Z Notation

Version 1.1

30th June 1995

Prepared by members of the Z Standards Panel

BSI Panel IST/5/-/19/2 (Z Notation)
ISO Panel JTC1/SC22/WG19 (Rapporteur Group for Z)

Project editor: John Nicholls

This is the first issue of a major revision of the Z Standard, replacing Version 1.0 and interim versions numbered 1.0x, and is circulated for review by the Z Standards Panel.

Virtually all sections of the Standard have been revised. When this version has been reviewed and updated, it will be given wider distribution and prepared for submission as an ISO Committee Draft.

Published in the United Kingdom

Document history:

Version 0.1 (19th March 1991): First Version of ZIP Deliverable D1.3.1. Distributed for review and approval by ZIP Standards Review Committee.

Version 0.2 (15th May 1991): Prepared for a meeting of the ZIP Standards Review Committee on 10th May, 1991. Approved for public issue as the current Z Base Standard. Incorporates minor changes to version 0.1.

Version 0.3 (25th November 1991): Produced for a meeting of the Standards Review Committee in December 1991 and incorporating many changes proposed and decided by the committee.

Version 0.4 (9th December 1991): Substantially the same as Version 0.3, incorporating corrections to minor typographical errors.

Version 0.5 (20th March 1992): Revision and extension of Version 0.4, with new introductory sections, an extensive revision of the definition of *expression*, and many other corrections and improvements.

Version 0.6 (21st October 1992): Major revision, in which sections containing the language description are restructured and updated, and all other sections revised.

Version 1.0 (30th November 1992): A version prepared for distribution at the 7th Z User Meeting and subsequent general distribution, incorporating changes made by the Z Standards Review Committee. Adopted as Working Draft of the Z Standards Panel.

Version 1.0x: Version numbers of this form denote revisions of the Working Draft, updates of Version 1.0.

Acknowledgement. Preparation of earlier versions of this document was supported as part of the ZIP project. *ZIP — A unification initiative for Z Standards, Methods and Tools* was partially funded by the Department of Trade and Industry and the Science and Engineering Council under their joint Information Engineering Advanced Technology Programme.

Editor's note: Further acknowledgements may be added here.

Z Notation

Version 1.1

30th June 1995

Prepared by members of the Z Standards Panel

BSI Panel IST/5/-/19/2 (Z Notation)
ISO Panel JTC1/SC22/WG19 (Rapporteur Group for Z)

Project editor: John Nicholls

This is the first issue of a major revision of the Z Standard, replacing Version 1.0 and interim versions numbered 1.0x, and is circulated for review by the Z Standards Panel.

Virtually all sections of the Standard have been revised. When this version has been reviewed and updated, it will be given wider distribution and prepared for submission as an ISO Committee Draft.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	30 June 1995	Final

4. TITLE AND SUBTITLE

Z-Notation V1.1

5. FUNDING NUMBERS

Unknown

6. AUTHOR(S)

Z Standards Panel Members, BSI, Panel, ISO Panel
Editor: John Nicholls

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Oxford University Computing Laboratory
Wolfson Building
Parks Road
Oxford OX1 3QD

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES

EOARD
PSC 802 Box 14
FPO 09499-0200

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

SPC 95-4004

11. SUPPLEMENTARY NOTES

Minutes are appendices. Use the letter as the first page of "Z-Notation V1.1".

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release: distribution is unlimited.

12b. DISTRIBUTION CODE

A

ABSTRACT (Maximum 200 words)

This is the first issue of a major revision of the Z Standard, replacing Version 1.0 interim versions numbered 1.0x, and is circulated for review by the Z Standards Panel. Virtually all sections of the Standard have been revised. When this version has been reviewed and updated, it will be given wider distribution and prepared for submission as an ISO Committee Draft.

14. SUBJECT TERMS

EOARD, Standard, Specification, Notation

15. NUMBER OF PAGES

227

16. PRICE CODE

N/A

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

OXFORD UNIVERSITY COMPUTING LABORATORY
Programming Research Group

James Martin Professor of Computing: C. A. R. Hoare FRS
Professor of Computing Science J. A. Goguen

Tel: +44 1865 273840
Tel: +44 1865 283504

95-4004

10 August 1995

Major Michael S. Markow
European Office of Aerospace
Research & Development
223/231 Old Marylebone Road
London UK NW1 5TH

Dear Michael,

I enclose a copy of Version 1.1 of the Z Standard, as part of the deliverables of the contract. I also include a copy of the minutes of our most recent meeting, giving an idea of the current status of the project.

For more information and advice on the progress of the work, please contact Randolph Johnson, whose email address is drj@tycho.ncsc.mil.

Yours sincerely,

John Nicholls

John Nicholls

Encl: Version 1.1 of Z Notation
Minutes of Meeting 28

cc: Randolph Johnson, Mike Field

Wolfson Building
Parks Road
Oxford. OX1 3QD

Enquiries: +44 1865 273838
FAX: +44 1865 273839
or +44 1865 273819

Email: *carh*@comlab.ox.ac.uk
goguen@comlab.ox.ac.uk

AQ F01-11-2350

Contents

Foreword

Notes on this section of the Z Standard

Section title: Foreword
Section editor: John Nicholls
Source file: part of front.tex
Most recent update: 7th June 1995
Formatted: 3rd July 1995

This document

This is the current version of the Z Standard being developed as a BSI and ISO standard. It is the Working Draft (WD) of the Z Standards Panel, BSI Panel IST/5/-/19/2: *Z Notation*), ISO Panel SC22/WG19 (Rapporteur Group for Z).

Document status

Some sections of this document have been revised and others are under review. As a consequence, this version of the standard is neither complete nor internally consistent. It has been prepared and given limited distribution in this form so that those working on its revision can provide commments for its improvement.

Comments on this document

Comments may be sent to

John Nicholls, Convener Z Standards Panel
Oxford University Computing Laboratory
Programming Research Group
Wolfson Building
Parks Road, Oxford OX1 3QD
United Kingdom.

Contributors

Note: The following lists are provisional

This version of the Z Base Standard has been written and edited as follows:

Project editor:	John Nicholls
Section authors and editors:	Stephen Brien Randolph Johnson Trevor King Peter Lupton John Nicholls Susan Stepney Jim Woodcock John Wordsworth ...
Additional contributions by:	Rob Arthan Paul Gardiner Roger Jones Jon Hall Will Harwood Ian Hayes Steve King Chris Sennett Ib Sørensen Pete Steggles Sam Valentine ...

Other contributors

The Z notation and its mathematical foundations have been developed by many people. A selected list of papers tracing a history of Z development is included in the *References* at the end of this document (see page 220). In addition to the listed contributors to the mathematical foundations of Z, many programmers, systems designers and architects have provided support by using Z and giving feedback to the designers of the notation.

Z Standards Panel

Development of the Z Base Standard has benefited from contributions and reviews by the *Z Standards Panel*, previously known as the *Z Standards Review Committee.*

The following list shows the current members of the Z Standards Panel, together with their alternates or deputies.

Derek Andrews	University of Leicester UK
Rob Arthan	ICL Winnersh UK
Peter Baumann	University of Zurich Switzerland
Cinzia Bonini	INTECS Pisa Italy
Stephen Brien	Oxford University PRG UK
Colin Champion	CESG Cheltenham UK
John Dawes	ICL Reading UK
Susan Gerhart	University of Houston USA
Jon Hall	University of York UK
Jonathan Hammond	Praxis plc Bath UK
Will Harwood	Imperial Software Technology Cambridge UK
Ian Hayes	University of Queensland Australia
Kees van Hee	Eindhoven University of Technology Netherlands
Randolph Johnson	DoD Fort Meade USA
Roger Jones	ICL Winnersh UK
Steve King	Oxford University PRG UK
Trevor King	Praxis plc Bath UK (up to Meeting 27)
Peter Lupton	IBM United Kingdom Laboratories Hursley UK
Peter Mataga	AT & T Bell Laboratories Naperville USA
Silvio Meira	University of Pernambuco Brazil
Akira Nakamura	Toshiba Corporation Kawasaki Japan
John Nicholls	Oxford University PRG UK *(Convener)*
Colin Parker	British Aerospace Warton UK
Jan Peleska	DST Kiel Germany
Brian Ritchie	Rutherford Appleton Laboratories Chilton UK
Gordon Rose	University of Queensland Australia
Mark Saaltink	ORA Ottowa Canada
Mayer Schwartz	Tektronix Oregon USA
Jane Sinclair	Open University UK
Alf Smith	DRA Malvern UK
Makoto Someya	Unisys Nihon Tokyo Japan
Pete Steggles	Imperial Software Technology Cambridge UK
Susan Stepney	Logica Cambridge UK
Mitsukazu Uchiyama	FDT-SWG Japan
Sam Valentine	Brighton University UK
Jim Woodcock	Oxford University PRG UK
John Wordsworth	IBM United Kingdom Laboratories Hursley UK
Pete Young	British Telecom UK

□

0 Introduction

Notes on this section of the Z Standard

Section title: Introduction
Section editor: John Nicholls
Source file: intro.tex
Most recent update: 16th jan 95
Formatted: 3rd July 1995

Z was originally developed as a *specification* notation for preparing formal descriptions of systems, without necessarily indicating how they will be implemented. This section includes a description of the aims and objectives of formal specification notations, with special reference to Z. The design principles used in the development of the Z standard are described.

0.1 Notations for system description

It is widely acknowledged that natural languages and similar informal notations have many disadvantages when used for writing technical descriptions. In using such languages it is difficult to write specifications with the required precision, clarity and economy of expression and to transform them systematically and reliably into code or hardware. Furthermore, it is impossible to carry out formal mathematical reasoning about informally written descriptions.

In contrast, specifications written in *formal* notations can be made precise and clear. Inference rules derived from their mathematical foundations enable designers to carry out mathematical reasoning and construct proofs relating to the properties of system descriptions.

The advantages of formal notations were recognised from an early stage in the history of computing, although it has taken considerable time for their practical application to become established. Many of the early large-scale applications of formal notation were for the specification of programming languages; formal descriptions of syntax are now widespread and for some languages there are formal descriptions of semantics.

Formal notations are now being used in a wide and expanding variety of environments, especially in key areas where the integrity of systems is critical, or where there is high intensity of use. For a discussion of domains of application for formal methods, see [19].

Examples of the effective use of formal specification notations are found in the following areas:

safety critical systems
security systems
the definition of standards
hardware development
operating systems
transaction processing systems

Descriptions of case studies from these and other application areas for Z are listed in a *Z Bibliography* by Bowen [2].

0.2 Objectives of a specification notation

The objectives of a formal specification notation are to assist in the production of descriptions that are complete, consistent and unambiguous. To achieve these objectives, a formal specification notation needs to be:

usable by those who read and write formal documents;

expressive, so that it can be used for a wide range of applications;

precise, so that it is possible to write descriptions that mean exactly what is intended;

given a *mathematically sound* meaning, since mathematical reasoning may be used in the development process;

suitable for defining sufficiently *abstract* models of systems that specifications do not need to contain unnecessary implementation details.

0.3 Characteristics of Z

A central part of Z is taken from the mathematics of set theory and first order predicate calculus. For the purposes of system description additions have been made to conventional mathematics, including:

a *type system* which requires each variable to be associated with a declared type. The ability to type-check a specification helps in assuring that it is accurate and consistent;

the *Z schema notation*, which provides a technique for grouping together and re-using common forms;

a *deductive system* which supports reasoning about Z specifications.

In addition, the following have been developed to help in the pragmatic use of Z in development projects:

the capability for writing explanatory text as an integral part of a Z document.

the inclusion within the standard of an agreed method of representing text in computers and transmitting it.

0.4 Design principles

The following design principles have been used in the development of the standard and are based on those used, explicitly or implicitly, in the original design of Z.

Basis in mathematics. Z is based on a central core of mathematics and uses accepted mathematical concepts and notation. In addition, there are means of defining and checking the *types* of Z elements and, by means of the Z *schema*, for structuring specifications.

Utility. All parts of Z included in the standard will have been shown to contribute to the main objectives of Z and will have been used in significant case studies or development projects.

Simplicity. There is an objective to keep the Z notation as simple as possible, consistent with its overall objectives.

0.5 Aims of standardisation

The Z standard supports the following general aims of standardisation as listed in the British Standards Institution *Standard for Standards* [5]:

provision of a medium for communication and interchangeability;

support for the economic production of standardised products and services;

the establishment of means for ensuring consistent quality and fitness for purpose of goods and services;

promotion of international trade.

0.6 Validation of the standard

In order to *validate* the standard, it is necessary to ensure that it is is appropriate, consistent and complete, and is in accordance with the general understanding of the Z notation. In order to achieve this, the following steps have been taken:

existing descriptions of the notation have been used as a basis for the document;

alternative concepts and notations have been proposed where existing ones were considered deficient;

the standard is being reviewed by the *Z Standards Review Committee*, which includes experts in formal methods, users and tool makers;

the standard is being reviewed by the ZIP tools project to confirm that it can be supported by tools;

the mathematical part of the standard is being checked for soundness.

□

1 Scope

Notes on this section of the Z Standard

Section title: Scope
Section editor: John Nicholls
Contributions by:
Source file: scope.tex
Most recent update: 16 January 1995
Formatted: 3rd July 1995

The Z standard defines the representation, structure and meaning of the formal part of specifications written in the Z notation.

In addition to defining the formal part of the Z notation, the Z standard defines:

a Library or Toolkit of mathematical functions for use in writing Z specifications;

an Interchange Format for Z documents that enables them to be prepared, stored and transmitted within computer networks;

a deductive system for formal reasoning about Z specifications.

A Z document may contain both formal and informal text. The lexis of the standard does not define how the formal and informal parts are delimited; this is defined in the Interchange Format. The Interchange Format does not define the structure of the informal part of a Z document.

The standard does not define a method of using Z.

□

2 Normative references

Notes on this section of the Z Standard

Section title: Normative references
Section editor: John Nicholls
Source file: normref.tex
Most recent update: 29th June 1995
Formatted: 3rd July 1995

BSI6154 BSI Standard **BS 6154**, *Method of defining syntactic metalanguage*, British Standards Institution, 1981.

ISO8879 ISO (International Organization for Standardization), **ISO 8879-1986 (E)** *Information Processing – Text and Office systems – Standard Generalized Markup Language (SGML)*, Geneva: ISO, 1986.

□

3 Conformity

Notes on this section of the Z Standard

Section title: Conformity
Section editor: John Nicholls
Source file: conform.tex
Most recent update: 16 January 1995
Formatted: 3rd July 1995

A specification conforms to the standard for the Z notation if and only if the formal text is written in accordance with the syntax rules and is well typed.

A deductive system for Z conforms to the standard if and only if its rules are sound with respect to the semantics.

□

4 Semantic metalanguage

Notes on this section of the Z Standard

Section title: Semantic metalanguage
Section editor: Randolph Johnson
Contributions by: Stephen Brien, Randolph Johnson, John Nicholls, Jim Woodcock, ... *(others to be added)*
Updated by: Randolph Johnson
Source file: math.tex
Most recent update: 29th June 1995
Formatted: 3rd July 1995

4.1 Introduction

This is the first of two chapters describing the mathematical framework used in the definition of Z. The chapter includes:

the names of metalanguage symbols;

the forms in which they are used;

descriptions of their meaning.

Many of the symbols used in this chapter are derived from conventional mathematics and are defined informally. Throughout the standard, the mathematical treatment is based on Zermelo-Fraenkel (ZF) set theory. An introduction to ZF theory can be found in text books such as Enderton [7] or Hamilton [10].

In addition to conventional mathematical symbols, special symbols are introduced which allow concise semantic definitions to be written. Where these have meanings similar to those of Z, Z-like symbols are used. Definitions of new symbols are given in terms of basic symbols (or other new symbols).

Note that, although symbols similar to those of Z are used, the semantic metalanguage is not Z but mathematics based on classical (i.e. untyped) set theory.

Naming conventions. The following naming conventions are used:

upper-case letters A, B, C (sometimes with subscripts) denote sets;

upper-case letters R, S, T (sometimes with subscripts) denote relations;

lower-case letters a, b, c (sometimes with subscripts) denote members of sets (which may also be sets themselves).

Commuting diagrams. In several of the following descriptions *commuting diagrams* are used to illustrate relationships between the set constructors being defined. Commuting diagrams are graphs whose nodes are labelled with sets. Nodes are connected by arrows, each arrow being labelled with a relation between the sets at each end. A diagram is said to *commute* when any two composed routes between nodes yield the same result.

Elision An expression such as $a_1, \ldots, a_n$ indicates that intermediate members in the finite list are elided. Whenever such an expression is used, the minimum value that n is allowed to take (usually 0 or 1) will be stated. An expression such as $A^1 \cup A^2 \cup \ldots$ indicates that all (finite) values of the superscript are to be included.

4.2 Definitions and declarations

Variables and notations are introduced and named as follows:

Table 1: Declarations and definitions

Name	Symbol	Example	Description
declaration	:	$a : A$	a is declared to be a member of the set A
definition	$\widehat{=}$	$A \mathrel{\widehat{=}} B$	A is defined as B

Note: A declaration of a in the form $a : A$ introduces the object a and states that it is a member of the set A. Care has been taken to ensure that when such a declaration is used, the set A is non-empty.

4.3 Sets

The following sets are predefined:

Table 2: Predefined sets

Name	Form	Description
empty set	$\varnothing$	the set having no members
integers	$\mathbb{Z}$	$\ldots, -2, -1, 0, 1, 2, \ldots$
strings	$\mathbb{S}$	the set of all finite strings of characters

Relationships between sets and their members are written as follows:

Table 3: Relationships between sets and members

Name	Form	Description
membership	$a \in A$	a is a member of A
subset	$A \subseteq B$	A is a subset of B i.e. all members of A are members of B
equality	$A = B$	sets A and B are equal i.e. A and B have the same members

4.3.1 Set constructors

The following *set constructors* define sets constructed from elements or from other sets:

Table 4: Set constructors

Name	Form	Description
set extension	$\{a_1, \ldots, a_n\}$	the set comprising $a_1, \ldots, a_n$; if $n = 0$, the set is the empty set; if $n = 1$, the set is a *singleton* set
union	$A \cup B$	the set comprising all the members of A and all the members of B
intersection	$A \cap B$	the set comprising the members common to A and B
set difference	$A \setminus B$	the set comprising the members of A that are not members of B
power set	$\mathbb{P} A$	the set of all subsets of A
finite power set	$\mathbb{F} A$	the set of all finite subsets of A

4.4 Tuples and products

The following constructors define tuples and products:

Table 5: Tuples and products

Name	Form	Definition
tuple	$(a_1, \ldots, a_n)$	ordered list of the elements $a_1, \ldots, a_n$, where $n \geq 1$
Cartesian product	$A_1 \times \ldots \times A_n$	the set of tuples $(a_1, \ldots, a_n)$ such that $a_1 \in A_1$ and ... and $a_n \in A_n$
enumerated product	A^n	the set of tuples $(a_1, \ldots, a_n)$ such that $a_1, \ldots, a_n \in A$
iterated product	A^+	$A^1 \cup A^2 \cup A^3 \cup \ldots$

Note: It is important in the metalanguage that the sets $\{A^1, A^2, A^3, \ldots\}$ form a disjoint collection. Under the most common construction of tuples, this need not be the case. However, it is true if, for example, we view a tuple as a finite sequence so that $(a_1, \ldots, a_n)$ is a function from $\{1, 2, \ldots, n\}$ into A.

4.5 Relations

The following are defined:

Table 6: Relations

Name	Form	Definition
relations	$A \leftrightarrow B$	$\mathbb{P}(A \times B)$
identity relation	I_A	$(a, b) \in I_A \Leftrightarrow a = b \land a \in A$
domain	$\mathrm{dom}\, R$	$a \in \mathrm{dom}\, R \Leftrightarrow \exists b \bullet (a, b) \in R$
range	$\mathrm{ran}\, R$	$b \in \mathrm{ran}\, R \Leftrightarrow \exists a \bullet (a, b) \in R$
converse	R^{-1}	$(a, b) \in R^{-1} \Leftrightarrow (b, a) \in R$
composition	$R \fatsemi S$	$(a, b) \in R \fatsemi S \Leftrightarrow$ $\exists c \bullet (a, c) \in R \land (c, b) \in S$
range restriction	$R \rhd A$	$R \fatsemi I_A$
range anti-restriction	$R \rhd\!\!\!\!- A$	$R \rhd (\mathrm{ran}\, R \setminus A)$
domain restriction	$A \lhd R$	$I_A \fatsemi R$
domain anti-restriction	$A \lhd\!\!\!\!- R$	$(\mathrm{dom}\, R \setminus A) \lhd R$

Note: The composition operator binds more tightly than the set constructors.

4.6 Functions

A function is a relation with the property that to each element in its domain there corresponds exactly one element in its range.

Table 7: Functions

Name	Form	Description or definition
partial functions	$A \pfun B$	the set of functions from A into B whose domains are subsets of A
total functions	$A \fun B$	the set of functions from A into B whose domains are the whole of A
total injections	$A \rightarrowtail B$	the set of total functions from A into B which are one-to-one
total surjections	$A \twoheadrightarrow B$	the set of total functions from A into B whose ranges are B
bijections	$A \rightarrowtail\!\!\!\!\!\twoheadrightarrow B$	$A \rightarrowtail B \cap A \twoheadrightarrow B$
finite partial functions	$A \finpfun B$	$A \pfun B \cap \mathbb{F}(A \times B)$

Table 8: Function constructors

Name	Form	Description or definition
constant function	a°_A	maps all members in the set A to a $(b, a) \in a^\circ_A \Leftrightarrow b \in A$
relational image	$^{\exists}(R)$	$(A, B) \in {}^{\exists}(R) \Leftrightarrow B = \mathrm{ran}(A \lhd R)$
singleton image	$^{\wedge}(R)$	$(a, B) \in {}^{\wedge}(R) \Leftrightarrow B = \mathrm{ran}(\{a\} \lhd R)$

Note: The subscript A may be omitted if the domain can be determined from the context.

In the remainder of this section, the term *function*, when not otherwise specified, is taken to mean *partial* function.

4.7 Set constructors as relations

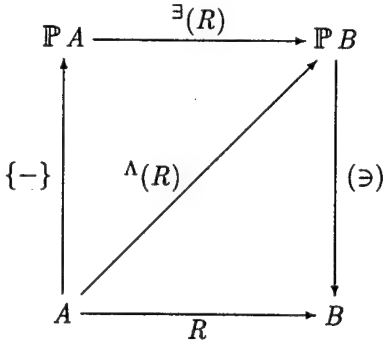
Membership, union, intersection etc. are not sets, and therefore, *a fortiori*, not relations. However, the restriction of any of these to the members or subsets etc. of a particular set A, does indeed yield a relation. In general, the relation determined, e.g., by the membership function, will depend on the set A; however, it is convenient to use a notation which supresses this dependence when A is clear from the context of use. The notations used are defined in the following table.

Table 9: Set constructors defined as relations

Name	Symbol	Domain	Range	Definition
union	$(\cup)$	$(\mathbb{P}\, A)^2$	$\mathbb{P}\, A$	$((a_1, a_2), b) \in (\cup) \Leftrightarrow b = a_1 \cup a_2$
intersection	$(\cap)$	$(\mathbb{P}\, A)^2$	$\mathbb{P}\, A$	$((a_1, a_2), b) \in (\cap) \Leftrightarrow b = a_1 \cap a_2$
set difference	$(\setminus)$	$(\mathbb{P}\, A)^2$	$\mathbb{P}\, A$	$((a_1, a_2), b) \in (\setminus) \Leftrightarrow b = a_1 \setminus a_2$
containment	$(\supseteq)$	$\mathbb{P}\, A$	$\mathbb{P}\, A$	$(a, b) \in (\supseteq) \Leftrightarrow b \subseteq a$
member	$(\ni)$	$\mathbb{P}\, A$	A	$(a, b) \in (\ni) \Leftrightarrow b \in a$
singleton set	$\{-\}$	A	$\mathbb{P}\, A$	$(a, b) \in \{-\} \Leftrightarrow b = \{a\}$
tuple set	$\{..\}$	A^+	$\mathbb{P}\, A$	$((a_1, \ldots, a_n), b) \in \{..\} \Leftrightarrow b = \{a_1, \ldots, a_n\}$
power	$(\mathbb{P})$	$\mathbb{P}\, A$	$\mathbb{P}\mathbb{P}\, A$	$(a, b) \in (\mathbb{P}) \Leftrightarrow b = \mathbb{P}\, a$
relational override	$(\oplus)$	$(A \leftrightarrow B)^2$	$A \leftrightarrow B$	$((R_1, R_2), S) \in (\oplus) \Leftrightarrow S = (\mathrm{dom}\, R_2 \lhd R_1) \cup R_2$
Cartesian product	$(\times)$	$(\mathbb{P}\, A)^+$	$\mathbb{P}(A^+)$	$((a_1, \ldots, a_n), b) \in (\times) \Leftrightarrow b = a_1 \times \ldots \times a_n$
indexed product	$\mathcal{X}$	$A \pfun \mathbb{P}\, B$	$\mathbb{P}(A \pfun B)$	$(a, b) \in (\mathcal{X}\, ; \ni) \Leftrightarrow b \subseteq a\, ; \ni$ where $\mathrm{dom}\, a = \mathrm{dom}\, b$.

These relations will be used only when they have well-defined domains.

The following diagram shows commuting properties of relational constructors:



4.8 Compatible functions.

Two functions are said to be *compatible* if their union is also a function. That is, their values agree whenever their domains overlap.

The set of pairs of compatible functions from A to B is defined as follows:

$$\mathcal{C}_{AB} \mathrel{\widehat=} \mathrm{dom}((\cup) \rhd (A \pfun B))$$

The functional forms of the set operators: union, intersection and set difference are defined only when the arguments are compatible functions. When defined, they have the same value as their set equivalents.

Table 10: Constructors on compatible functions

Name	Symbol	Definition
functional union	$\sqcup_{AB}$	$\mathcal{C}_{AB} \lhd (\cup)$
functional intersection	$\sqcap_{AB}$	$\mathcal{C}_{AB} \lhd (\cap)$
functional difference	$\leftharpoondown_{AB}$	$\mathcal{C}_{AB} \lhd (\setminus)$

4.9 Diagonal and projection

The following are defined:

Table 11: Diagonal and projection operators

Name	Symbol	Domain	Range	Definition
diagonal	Δ_n	A	A^n	$(a, (a_1, \ldots, a_n)) \in \Delta_n$ $\Leftrightarrow$ $a_1 = a \land \ldots \land a_n = a$, where $n \geq 1$
projection	π_i	$A_1 \times \ldots \times A_i \times \ldots \times A_n$	A_i	$((a_1, \ldots, a_i, \ldots, a_n), a) \in \pi_i$ $\Leftrightarrow$ $a = a_i$, where $1 \leq i \leq n$

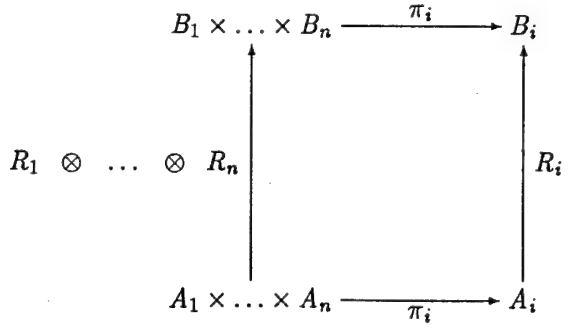
4.10 Pointwise product

The pointwise product $R_1 \otimes \ldots \otimes R_n$ is a relation from the Cartesian product of the domains of $R_1, \ldots, R_n$ to the Cartesian product of their ranges.

Table 12: Pointwise product constructors

Name	Form	Definition
pointwise product	$R_1 \otimes \ldots \otimes R_n$	$((a_1, \ldots, a_n), (b_1, \ldots, b_n)) \in R_1 \otimes \ldots \otimes R_n$ $\Leftrightarrow$ $(a_1, b_1) \in R_1 \land \ldots \land (a_n, b_n) \in R_n$
iterated pointwise product	$R^{\oplus}$	$R \cup (R \otimes R) \cup (R \otimes R \otimes R) \cup \ldots$

The following diagram illustrates properties of the product constructors:



4.11 Relational tuple

If A is the intersection of the domains of the relations $R_1, \ldots, R_n$, then the relational tuple $\langle R_1, \ldots, R_n \rangle$ is a relation from A to the Cartesian product of their ranges. This is defined in terms of the relational product operator and the diagonal operator Δ_n.

Table 13: Tuple constructor

Name	Form	Definition
relational tuple	$\langle R_1, \ldots, R_n \rangle$	$\Delta_n \fatsemi (R_1 \otimes \ldots \otimes R_n)$, where $n \geq 1$

If the relations $R_1, \ldots, R_n$ each have domain A and have range B_i, respectively, then the following diagram shows the relationship between relational tuple $\langle R_1, \ldots, R_n \rangle$ and projection:

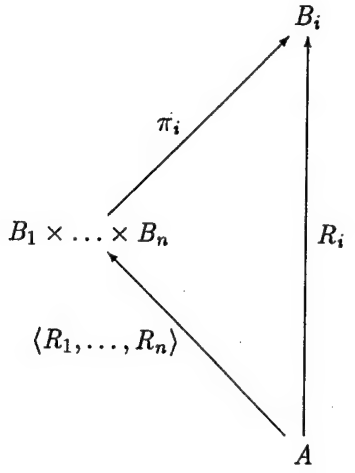


Table 14: Relational tuple operator

Name	Symbol	Domain	Range	Definition
relational tupling	$\diamond$	$(A \leftrightarrow B_1) \times (A \leftrightarrow B_2)$	$A \leftrightarrow (B_1 \times B_2)$	$((R_1, R_2), S) \in \diamond$ $\Leftrightarrow S = \langle R_1, R_2 \rangle$

4.12 Promoted application

In order to avoid generating potentially undefined terms, there is no function application in the meta-language (it is used only in explanatory notes). For the definition of function application in Z, it is necessary to define a form of promoted application. For any given a, the apply-to-a function takes as its argument a function and has as its result the application of that function to the element a. This function can be generalised to the *promoted application* operator $(R \bullet T)$, which is the relational analogue of the S combinator in combinatory logic.

Table 15: Promoted application

Name	Form	Domain	Range	Definition
apply-to-a	$(\_a)$	$A \leftrightarrow B$	B	$((\ni) \cap (a^\circ \fatsemi \pi_1^{-1})) \fatsemi \pi_2$
promoted application	$R \bullet S$	$A \leftrightarrow (B \leftrightarrow C) \times (A \leftrightarrow B)$	$A \leftrightarrow C$	$((R \fatsemi \ni) \cap (S \fatsemi \pi_1^{-1})) \fatsemi \pi_2$

Note: If ρ is a function and c is an member of the domain of ρ then the following equality holds: $(\_c)\,\rho \;=\; \rho\, c$. So we have the following equivalence:

$$(a, b) \in (\_c) \;\Leftrightarrow\; (c, b) \in a$$

If the relations $R : A \leftrightarrow (B \leftrightarrow C)$ and $S : A \leftrightarrow B$ both have the element a in their domains, then the tuple $(a, (b, c))$ belongs to $(R \fatsemi \ni)$ providing that (b, c) is a member of the set $R(a)$ and it belongs to $(S \fatsemi \pi_1^{-1})$ if b is $S(a)$. The tuple (a, c) belongs to the whole relation exactly when for some b the tuple $(a, (b, c))$ belongs to the first part. If R and S are functions, then promoted application is defined so that the following equality holds:

$$(R \bullet S)(a) \;=\; (R\, a)(S\, a)$$

Promoted application is disjunctive in both arguments.

The apply-to-a function $(\_a)$ can be derived from promoted application as follows:

$$(\_a) \;=\; I \bullet a^\circ$$

□

5 Semantic universe

Notes on this section of the Z Standard

Section title: Semantic universe
Section editor: Randolph Johnson (ex: Jim Woodcock)
Contributions by: Stephen Brien, Jim Woodcock, ... *(others to be added)*
Source file: semdom.tex
Most recent update: 29th June 1995
Formatted: 3rd July 1995

5.1 Introduction

This section defines a semantic universe within which the meanings of Z specifications lie; it is based on the Zermelo-Fraenkel axiomatisation of sets mentioned in the last section.

The syntax of Z defines a set of specifications. The semantics of Z defines a function from these specifications to meanings within the semantic universe. This universe contains the meanings of names, types, and values used in a specification, as well as the environment used to define the overall meaning of a specification. It should be noted that the meaning of a specification is further paramatrised by the assignment of sets to given set names in the specification.

5.2 Names and types

The first task in building the universe is to explain the use of names and the notion of types. In Z, a name is used to denote an element, which may be a set, a tuple, a binding, or an element of a given type. These names come in three varieties: they may be the names of schemas, variables, or constants. This partitioning of abstract names is dependent on the specification in question, the members of each set not being distinguishable in the concrete syntax. Abstractly, we have that, for any particular Z specification, our set of names, *Name*, is comprised of schema names, variable names, and constant names:

$$SchemaName \cup Variable \cup Constant = Name$$

Representation names can have different abstract forms for different specifications; there is assumed to be an infinite supply of each.

In common with other specification and programming languages, but unlike ZF set theory, the rules of Z require that every name introduced in a Z specification is given a particular type which determines the possibilities for the values that it may take.

The simplest types are *given set names*, which are used to introduce abstract objects into a specification, as the formal names of generic parameters or as expressions. Their names are drawn from the set *Constant*.

$$GivenSetName \subseteq Constant$$

Note: The names $\mathbb{Z}$ for the set of integers and $\mathbb{S}$ for the set of strings are members of the set of given set names. I.e., $\{\mathbb{Z}, \mathbb{S}\} \subseteq \mathit{GivenSetName}$.

Every type belongs to the set *Type*, which is partitioned into the four subsets *Gtype*, *Ptype*, *Ctype*, and *Stype* representing the given types, power set types, Cartesian product types, and schema types, respectively.

Basic familiarity with elementary set theory leads one to view something of given type as an object, of power set type as a set, of Cartesian product type as a tuple, but what about something of schema type? It is a partial function from variable names to types; such a function is called a signature:

$$\mathit{Signature} \mathrel{\widehat=} \mathit{Variable} \pfun \mathit{Type}$$

Now we have everything that we need in order to explain the structure of the set of types. Consider power set types. From every type represented by σ, we can construct the unique type which is represented by $\mathbb{P}\,\sigma$; every power set type is constructed in this way from a unique type. Thus, the power set type constructor is a *bijection* between *Type* and *Ptype*. Similar arguments apply to the other type constructors. We can sum this up by defining the following four bijections with the partitions of *Type*:

$$\begin{aligned}
&\mathit{givenT} : \mathit{GivenSetName} \rightarrowtail\!\!\!\!\!\twoheadrightarrow \mathit{Gtype} \\
&\mathit{powerT} : \mathit{Type} \rightarrowtail\!\!\!\!\!\twoheadrightarrow \mathit{Ptype} \\
&\mathit{cproductT} : (\mathit{Type}^+ \setminus \mathit{Type}) \rightarrowtail\!\!\!\!\!\twoheadrightarrow \mathit{Ctype} \\
&\mathit{schemaT} : \mathit{Signature} \rightarrowtail\!\!\!\!\!\twoheadrightarrow \mathit{Stype}
\end{aligned}$$

Note: The signature parameter for the *schemaT* operator can be the empty signature.

For each specification there is a set of distinct given types. All other types used are constructed from these given types using a *unique* combination of the type constructors. This uniqueness is guaranteed because the type constructors are in bijection with the partitions of the set *Type*. Therefore the set *Type* is the smallest set which is closed under these type constructors. In terminology from category theory, *Type* can be described as the initial algebra over the signature given by *givenT*, *powerT*, *cproductT*, *schemaT*. Using the notation for free types as defined in Z, we can sum this up by defining the set *Type* as follows:

$$\begin{aligned}
\mathit{Type} ::= &\ \mathit{givenT} \langle\!\langle \mathit{GivenSetName} \rangle\!\rangle \\
\mid &\ \mathit{powerT} \langle\!\langle \mathit{Type} \rangle\!\rangle \\
\mid &\ \mathit{cproductT} \langle\!\langle (\mathit{Type}^+ \setminus \mathit{Type}) \rangle\!\rangle \\
\mid &\ \mathit{schemaT} \langle\!\langle \mathit{Signature} \rangle\!\rangle
\end{aligned}$$

5.3 Values in Z

One of the purposes of ascribing a type to a variable is to determine which values the variable may take. To make this possible, each type has a (ZF) set of values associated with it, called its *carrier set*. The values in the carrier set of a given type are regarded as atomic objects. Each value in the carrier set of a non given type is modelled by a ZF set. The relationship between the types and values in a specification is defined by the function *Carrier*, whose definition we approach inductively by defining the carrier function for given types and then constructing the function for other types from this.

Note: In Z a type is represented by its carrier set.

A set W will be defined below to contain the values of all elements in Z. The carrier sets for each type in Z are subsets of W. The set $\mathcal{W}_0$ is the set comprising the carrier sets for each of the given types.

Definition 5.1 *For each specification there is a carrier function which maps the given types to elements of* $\mathcal{W}_0$.

$$Carrier_0 : Gtype \longrightarrow \mathcal{W}_0$$

Note: The carrier sets (elements of $\mathcal{W}_0$) may be empty sets. This means that the types are not inhabited.

Note: Suppose that γ is a given type; what is the carrier set of the power set type $powerT\,\gamma$? It is simply the set $\mathbb{P}(Carrier\,\gamma)$. In general, if γ is a power set type of a given type T, we must calculate the carrier set by stripping off the power set constructor, calculating the carrier set of this underlying given type, and then forming the power set of the result. This is given by the expression

$$(powerT^{-1} \fatsemi Carrier_0 \fatsemi (\mathbb{P}))\,\gamma$$

Similarly, if γ is a Cartesian product of given types, then we should break it up into its constituent given types, determine their carrier sets, and then form their Cartesian product, so that we end up with a set of tuple values:

$$(cproductT^{-1} \fatsemi Carrier_0^{\oplus} \fatsemi (\times))\,\gamma$$

Finally, if γ is a schema type constructed from given types, then we should obtain the underlying signature; this yields a function from variable names to types, which we must turn into a function from variable names to the carrier sets of these types; finally, we must form the schema product, so that we end up with a set of functions from names to values:

$$(schemaT^{-1} \fatsemi {}^{\exists}(I_{Variable} \otimes Carrier_0) \fatsemi \mathcal{X})\,\gamma$$

The indexed product operator $\mathcal{X}$ is used to convert a function $Variable \pfun \mathbb{P}\,W$ to a set of functions $\mathbb{P}(Variable \pfun W)$.

In this discussion, we have assumed that the type constructors are applied to given types, but in general they are applied to arbitrary types. Since a type is made out of a finite sequence of applications of the constructors, we can define the *depth* of a type to be the length of this sequence. Now we can give our inductive definition using this notion of depth:

Definition 5.2

$$
\begin{aligned}
&Carrier_{i+1} \mathrel{\widehat=} \\
&\quad Carrier_i \\
&\quad \cup\, powerT^{-1} \fatsemi Carrier_i \fatsemi (\mathbb{P}) \\
&\quad \cup\, cproductT^{-1} \fatsemi Carrier_i^{\oplus} \fatsemi (\times) \\
&\quad \cup\, schemaT^{-1} \fatsemi {}^{\exists}(I_{Variable} \otimes Carrier_i) \fatsemi \mathcal{X}
\end{aligned}
$$

Note: The carrier set of the schema type constructed from the empty signature is the set containing the empty binding.

In order to calculate the carrier set for a type γ, we must apply $Carrier_i$, where i is the depth of type γ. Notice that every carrier function whose domain contains γ gives the same result for γ; this justifies our general definition.

Definition 5.3 *The general carrier function mapping elements of Typeto their carrier sets is defined as follows:*

$$Carrier \mathrel{\widehat=} Carrier_0 \cup Carrier_1 \cup Carrier_2 \cup \ldots$$

The values which may be used in a Z specification are those that are in the carrier sets that are assigned to the types. This set is constructed from the elements of $\mathcal{W}_0$ using the type constructors.

Definition 5.4 *The set W of all values is the set of all the elements in each of the carrier sets for the elements of Type:*

$$W \mathrel{\widehat=} {}^{\exists}(Carrier \fatsemi \ni)\ Type$$

Definition 5.5 *A binding is a finite mapping from variables to values:*

$$Binding \mathrel{\widehat=} Variable \pfun W$$

The carrier function is a homomorphism between *Type* and W.

Note: Thus, we have the equations for carrier

$$
\begin{aligned}
&Carrier\ (powerT\ \gamma) = \mathbb{P}\,(Carrier\ \gamma) \\
&Carrier\ (cproductT\ (\gamma_1, \ldots, \gamma_n)) = (Carrier\ \gamma_1) \times \ldots \times (Carrier\ \gamma_n) \\
&Carrier\ (schemaT\ \gamma) = \mathcal{X}({}^{\exists}(I \otimes Carrier)\ \gamma)
\end{aligned}
$$

5.4 Elements in Z

Each element in Z is represented by the pair consisting of its type and its value. The semantic set *Elm* is a set of type-value pairs; this set may be considered as the relation between types and values in which a type is related to a value if and only if the value is a member of the carrier set of the type.

Editor's note: The commuting diagram "The type system" has been temporarily omitted from this section.

Definition 5.6 *The set Elmis a set of type-value pairs:*

$$Elm : \mathbb{P}(Type \times W)$$

A value is an element of a type if and only if it is contained in the carrier set of the type:

$$Elm \mathrel{\widehat=} Carrier \mathbin{;} \ni$$

Note: The set *Elm* of all compatible type-value pairs is also a relation between types and elements of their carrier sets. If the carrier set for a type is empty, then the type will not be in the domain of *Elm*.

The first and second projections on a tuple are used to extract the type and value respectively.

Definition 5.7 *The type and value functions are projections from the tuples in Elm:*

$$t \mathrel{\widehat=} Elm \lhd \pi_1$$
$$v \mathrel{\widehat=} Elm \lhd \pi_2$$

Sets in Z are those elements which have a power type:

Definition 5.8 *The set Pelmcontains all elements which have power type:*

$$Pelm \mathrel{\widehat=} Ptype \lhd Elm.$$

Definition 5.9 *The membership relation, $\exists$, for elements in Z is a relation between Pelm and Elm:*

$$\exists \; : Pelm \leftrightarrow Elm$$

This relation is the product of the relation between a power type and its underlying type and the relation between a set and its members:

$$\exists \mathrel{\widehat=} (powerT^{-1} \otimes \ni)$$

A Z specification consists of a number of definitions which introduce names. Each name may denote some value, and each name must have some type; that is, each name may be associated with an *element.* We call such an assignment of elements to names a *situation.*

Definition 5.10 *A situation is a finite mapping from variables to elements:*

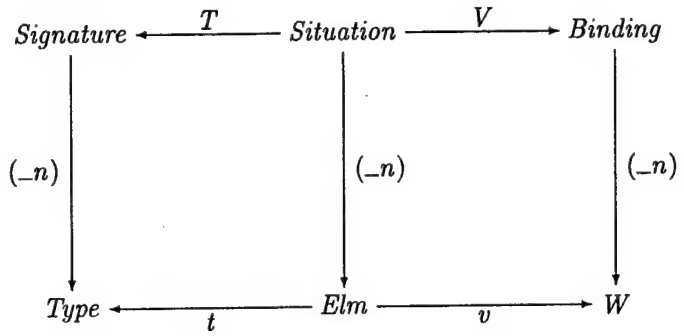
$$Situation \mathrel{\widehat=} Variable \pfun Elm$$

A situation tells us two things about the names in a specification: their types and their values. If we think about the type projection of each name, then we obtain a mapping from names to types: a signature. If, on the other hand, we think about the value projection of each name, then we obtain a mapping from names to values: a binding. The signature and binding corresponding to a particular situation can be extracted by the functions T and V respectively.

Definition 5.11 *The T and V functions are defined as follows:*

$$T \mathrel{\widehat=} {}^{3}(I_{Variable} \otimes t)$$
$$V \mathrel{\widehat=} {}^{3}(I_{Variable} \otimes v)$$

The following commuting diagram, in which n is an arbitrary variable name, illustrates the relationship between types and values and their lifted forms as signatures and bindings:



5.5 Generics

A Z expression that involves a generic instantiation acquires a type and a value that depends upon the type and value of the expression used in the instantiation. Thus if we see $\varnothing_{[\mathbb{Z}]}$, we know this has a different type from $\varnothing_{[\mathbb{P}\,\mathbb{Z}]}$. The various types that $\varnothing$ may take are represented as a *function* from *Type* to *Type*. In the case of $\varnothing$, this function takes an arbitrary powerset type to itself. In general, where a generic definition contains a list of identifiers, the various possible instantiations are a function from lists of elements to a type and value. The elements which may appear as actual parameters of a generic definition must be of powerset type.

5.5.1 Generic types

For each generic type the number of formal parameters is fixed, and every possible sequence of powerset types with the right number of formal parameters is given a type. So each generic type is a function from fixed-length sequences of power types to a type. In order to simplify the definition of generic types we consider first the case of the generic type with n parameters and then extend to the general case. This type is a function from an n-tuple of power types to a type. This function must be total as all possible combinations of parameters must be given a type.

Definition 5.12 *For any natural number $n > 0$, the set of all generic types with n parameters is defined as follows:*

$$Gen\_Type_n \mathrel{\widehat=} Ptype^n \longrightarrow Type$$

Since the number of parameters for a generic type is fixed, the general generic type is an example of one of the specific fixed length generic types. So the set of all generic types is the union of all the fixed length types.

Definition 5.13 *The set of all generic types is the union of all the sets of finite length generic types:*

$$Gen\_Type \mathrel{\widehat=} Gen\_Type_1 \cup Gen\_Type_2 \cup \ldots$$

Note: This models a set far bigger than that which can be constructed using generic definitions in Z

5.5.2 Generic elements

As with generic types, for each generic element there is a fixed number of formal parameters that it can take; furthermore every possible sequence of the correct number of elements with powerset type is given a type and value. Generic elements are defined in a similar way to generic types: by defining the specific n-length case and generalising.

For any natural number $n > 0$, the set of all generic elements with n parameters is a subset of the set of functions from n-tuples of set elements to elements:

$$Gen\_Elm_n : \mathbb{P}(Pelm^n \longrightarrow Elm)$$

The functions representing generic elements are type consistent; a generic element, when instantiated with two sequences of elements of the same type, will give two elements of the same type. This is an important restriction on the functions used to model generic elements. In order to define this property it is necessary to characterise the type part of a generic element.

Definition 5.14 *The function τ_n takes a function from n-tuples of elements to elements and returns a relation from n-tuples of type to type:*

$$\tau_n : (Pelm^n \longrightarrow Elm) \longrightarrow (Ptype^n \longleftrightarrow Type)$$

$$\tau_n \mathrel{\widehat=} {}^{\exists}(t^n \otimes t)$$

The functions which are to be characterised as generic elements are those whose type part is a generic type, i.e. those whose type part is functional.

Definition 5.15 *The set of generic elements with n parameters are those functions whose type part is functional, i.e. contained in Gen_Type_n:*

$$Gen\_Elm_n \mathrel{\widehat=} \mathrm{dom}\,(\tau_n \rhd Gen\_Type)$$

In the same way as for generic types, the general generic element is an example of a specific one for some fixed number of parameters.

Definition 5.16 *The set of all generic elements is the union of all the sets of finite length generic elements:*

$$Gen\_Elm \mathrel{\widehat=} Gen\_Elm_1 \cup Gen\_Elm_2 \cup \ldots$$

Definition 5.17 *The generic type function can be generalised as follows:*

$$\tau \mathrel{\widehat=} \tau_1 \cup \tau_2 \cup \ldots$$

5.6 Environments

In order to give a meaning to the constructs of Z, we need an environment to record the elements denoted by the names used in a Z specification. The meaning of a Z specification is a set of environments. This set contains those environments which map the names declared in the specification to a combination of values which correspond to the constraints within the specification.

Definition 5.18 *An environment is defined as a finite partial function from names to elements or generic elements:*

$$Env \mathrel{\widehat=} Name \pfun (Elm \cup Gen\_Elm)$$

Whether a Z specification is well typed or not is a question that is independent of the *values* of the declared variables. To be able to answer this question it is simply necessary to have an environment in which the types of all names are recorded.

Definition 5.19 *A type-environment is defined as a finite function from names to types or generic types:*

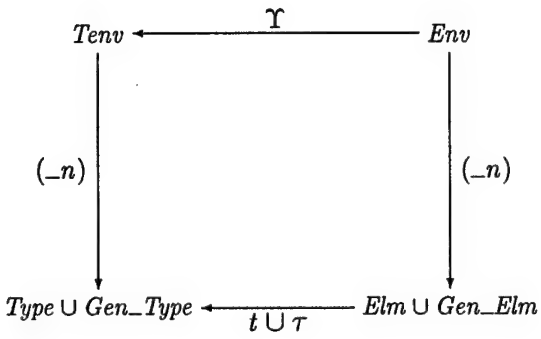
$$Tenv \mathrel{\widehat=} Name \pfun (Type \cup Gen\_Type)$$

The simple relationship between the richer environment, *Env*, and the one used just for type checking, *Tenv*, is given by the forgetful function Υ which 'throws away' the values.

Definition 5.20 *The function Υ maps the second element of each tuple in an environment onto its corresponding type or generic type:*

$$\Upsilon \mathrel{\widehat=} {}^{\exists}(I_{Name} \otimes (t \cup \tau))$$

The following commuting diagram, in which n is an arbitrary name, illustrates the relationship between the environment and the type-environment:



Note: If T is a set of type environments, then $^{\exists}(\Upsilon^{-1})\,T$ is the corresponding set of full environments.

□

6 Language description

Notes on this section of the Z Standard

Section title: Language description
Section editor: Randolph Johnson
Contributions by: ... *(to be added)*
Source file: lang.tex
Most recent update: 29th June 1995
Formatted: 3rd July 1995

6.1 Introduction

This section provides a general introduction to the following sections of the language definition, each of which defines a major syntactic category: *expression*, *predicate*, *schema*, *paragraph*, *specification*. Within each section there are subsections corresponding to the syntactic categories of the abstract syntax. These follow a consistent pattern, sub-divided as follows: *Abstract syntax*, *Concrete form*, *Sample representation and transformation*, *Type*, and *Value/Meaning*.

A *denotational* style of semantic description is used, as described for example in [25] and, as in the customary style of writing denotational semantics, semantic brackets are used to delimit text for which denotations are given. The notation is extended by providing different shapes of brackets for different kinds of language elements as shown in the following table. Three types of semantic functions are used, for *type*, *value* and *meaning*. The different types are identified by, respectively, the superscripts $\mathcal{T}, \mathcal{V}, \mathcal{M}$ on the brackets.

Table 16: Brackets used for semantic functions

Bracket	Argument	Forms
$[\![\_]\!]$	Expression	$[\![\_]\!]^{\mathcal{T}}$, $[\![\_]\!]^{\mathcal{V}}$, $[\![\_]\!]^{\mathcal{M}}$
$\{\!\| \_ \|\!\}$	Predicate	$\{\!\| \_ \|\!\}^{\mathcal{T}}$, $\{\!\| \_ \|\!\}^{\mathcal{V}}$, $\{\!\| \_ \|\!\}^{\mathcal{M}}$
$(\!\| \_ \|\!)$	Schema	$(\!\| \_ \|\!)^{\mathcal{T}}$, $(\!\| \_ \|\!)^{\mathcal{M}}$
$\langle\!\| \_ \|\!\rangle$	Paragraph	$\langle\!\| \_ \|\!\rangle^{\mathcal{T}}$, $\langle\!\| \_ \|\!\rangle^{\mathcal{M}}$
$?\_?$	Specification	$?\_?^{\mathcal{T}}$, $?\_?^{\mathcal{M}}$

The following meta-variables are used for expressing the *representation syntax*, i.e., the visual form in which it appears. Similar symbols, in bold font, are used for expressing the abstract syntax.

Table 17: Meta-variables used in the representation syntax

Variables	Sort
E, x, y	Expression
n, m	Name
a	String
i	Number
t	Tuple
s, u	Set-valued expression
b	Binding
f	Function
P, Q	Predicate
C, D	Declaration
St	Schema Text
S, T	Schema
Par	Paragraph

6.2 Abstract syntax

For each language element, its abstract syntax is defined in a form of BNF. The following example illustrates the style used.

POWERSET = **Pow** EXP

In some cases symbols such as θ are used rather than key-words or other structures in the syntax to make reading of the abstract syntax easier. The abstract syntax is presented in Annex A.

6.3 Concrete form, representation and transformation

Editor's note: This, and the next subsections, have been revised to account for the new concrete syntax and lexis.

Check carefully! JEN

For each language element, there is an example of the concrete form showing a production or productions of the language element being defined, together with a table showing the relationship between the representation and abstract forms.

> **Note:** There may be more than one representation of an abstract syntax category; in such cases all forms are listed. In some cases the multiplicity of representations is due to the fact that some forms can be considered as abbreviations of others.

Transformations are presented in a denotational style. Superscripts on brackets denote the type of the argument.

Table 18: Transformation functions

Brackets	Argument
$[\![\_]\!]^{\mathcal{E}}$	Expression
$[\![\_]\!]^{\mathcal{P}}$	Predicate
$[\![\_]\!]^{\mathcal{D}}$	Declaration
$[\![\_]\!]^{\mathcal{S}}$	Schema
$[\![\_]\!]^{\mathcal{PAR}}$	Paragraph

The following example illustrates how a sample form from the concrete syntax is presented, together with a metalanguage version of the representation and its corresponding abstract form:

Concrete form

```
PSET Expression
```

Sample representation and transformation

Representation	Abstract
$\mathbb{P}\, s$	$\mathbf{Pow}[\![s]\!]^{\mathcal{E}}$

In this example the production for power set shows how a power set is represented as an expression prefixed with the power set symbol. The first column in the table gives an example of the representation form. In this case s is some expression for a set in representational form. The second column gives the abstract form of this expression. In this case the form is an (abstract) powerset symbol, followed by the abstract form of the expression s.

These two columns can be read as an equation in the form:

$$[\![\mathbb{P}\, s]\!]^{\mathcal{E}} \quad = \quad \mathbf{Pow}\, [\![s]\!]^{\mathcal{E}}$$

The concrete syntax is presented in Annex B.

6.4 Type

The definition of the Z type system is by structural induction over the abstract representation of a Z specification. The well-typedness of a Z specification can be determined independently of the *values* of the declared variables. So we see that the following definition of the Z type system is entirely self-contained: given a Z specification, the type definitions determine whether that specification is well-typed.

Note: Determining whether a given specification is well-typed is a decidable question. Similarly, the determination of the type of any term, within a given environment, is decidable. This is in contrast with evaluation – determining whether a term has a certain value is, in general, undecidable.

Table 19: Type functions of major forms

Name	Form	Sort
Expression Type	$[\![\boldsymbol{E}]\!]^{\mathcal{T}}$	$\mathit{Tenv} \pfun \mathit{Type}$
Predicate Type	$\{\!\| \boldsymbol{P} \|\!\}^{\mathcal{T}}$	$\mathbb{P}\, \mathit{Tenv}$
Schema Type	$(\!\| \boldsymbol{S} \|\!)^{\mathcal{T}}$	$\mathit{Tenv} \pfun \mathit{Signature}$
Paragraph Type	$\langle\!\| \boldsymbol{Par} \|\!\rangle^{\mathcal{T}}$	$\mathit{Tenv} \pfun \mathit{Tenv}$

The following example illustrates the description of the type of a powerset:

Type The type of the power set **Pow** $\boldsymbol{s}$ is the power set type of the type of the set $\boldsymbol{s}$.

$$[\![\mathbf{Pow}\, \boldsymbol{s}]\!]^{\mathcal{T}} \quad = \quad ([\![\boldsymbol{s}]\!]^{\mathcal{T}} \rhd \mathit{Ptype}) \fatsemi \mathit{powerT}$$

Note: A power set **Pow** $\boldsymbol{s}$ is well typed only if $\boldsymbol{s}$ has power set type.

The type description contains an informal description, the mathematical definition of the type function for the powerset and an explanation of when it is well-typed. This last explanation is derived directly from the domain of the type function.

Table 20: Meaning functions of major forms

Name	Form	Sort
Expression Meaning	$[\![\boldsymbol{E}]\!]^{\mathcal{M}}$	$Env \pfun Elm$
Predicate Meaning	$\{\!\| \boldsymbol{P} \|\!\}^{\mathcal{M}}$	$\mathbb{P}\, Env$
Schema Meaning	$(\!\| \boldsymbol{S} \|\!)^{\mathcal{M}}$	$Env \rightarrow\!\!\!\!\!\!\rightarrow Situation$
Paragraph Meaning	$\langle\!\langle \boldsymbol{Par} \rangle\!\rangle^{\mathcal{M}}$	$Env \rightarrow\!\!\!\!\!\!\rightarrow Env$

6.5 Meaning

The meanings of *expression, predicate, schema* and *paragraph* are given by the functions in the following table:

> **Note:** There is a need to explain that the definition of *Env* is paramaterised by the assignement of values to the given sets.

The meanings of expression, predicate, and schema are combined to provide a meaning for a paragraph. This meaning is a relation between environments. The meaning of a specification is defined as the image of the empty environment through the composition of the relations for all the paragraphs in the specification.

> **Note:** Now that declarations are no longer in the abstract syntax, the example below should be replaced.

The following example illustrates the description of the meaning of a simple declaration:

> **Meaning** The meaning of a compound declaration is the set of situations that, when restricted to the alphabet of each component, satisfy that component:
>
> $$(\!\| \boldsymbol{S}_1 \,;\, \boldsymbol{S}_2 \|\!)^{\mathcal{M}} \;=\; \langle (\!\| \boldsymbol{S}_1 \|\!)^{\mathcal{M}}, (\!\| \boldsymbol{S}_2 \|\!)^{\mathcal{M}} \rangle \,;\, \sqcup.$$
>
> **Note:** A compound declaration $\boldsymbol{D}_1; \boldsymbol{D}_2$ is value-defined only if both the declarations $\boldsymbol{D}_1$ and $\boldsymbol{D}_2$ are value-defined and if repeated declarations are value compatible.

The meaning description contains an informal description, the mathematical definition of the meaning function for the declaration and an explanantion of when it is value-defined. This last explanation is derived directly from the domain of the meaning function.

The meanings of pairs of expressions, predicates, etc., can be compared. For example, two expressions e_1 and e_2 are said to be semantically equivalent, written $e_1 \equiv e_2$, when $[\![e_1]\!]^{\mathcal{M}} = [\![e_2]\!]^{\mathcal{M}}$.

6.6 Value

The meaning functions for expressions and predicates are defined in terms of their type and value. So the value functions are the primitives defined in the following sections and have the structure shown in the following table:

Table 21: Value functions of major forms

Name	Form	Sort
Expression Value	$[\![E]\!]^{\nu}$	$Env \pfun W$
Predicate Value	$\{\!\![P]\!\!\}^{\nu}$	$\mathbb{P}\, Env$

The following example illustrates the description of the value of a powerset:

Value The value of the power set **Pow** s is the set of all the subsets of the value of s:

$$[\![\mathbf{Pow}\, s]\!]^{\nu} = [\![s]\!]^{\nu} ; (\mathbb{P})$$

Note: A powerset **Pow** s is value-defined only if the expression s is value-defined.

The value description contains an informal description, the mathematical definition of the value function for the powerset and an explanation of when it is value-defined. This last explanation is derived directly from the domain of the value function.

Editor's note: The following subsections need to be revised and possibly moved. In Version 1.1, the discussion of free variables has been moved to Annex F, *The logical theory of Z.*

6.7 Free variables

Ordinarily the definition of the free variables of an expression can be considered as a function on the names of identifiers appearing in the text of the expression and the variables bound by the declarations. In Z however, the case is somewhat more complicated. The use of schema references as declarations means that there is an implicit declaration. The names introduced by the declaration S where S is a schema reference are related not to the name S but to its value in the particular environment within which it is being evaluated. In other words the free variables of an expression depend on the text of the expression *and* the environment in which the expression is evaluated.

We define the *free variables* of an expression to be a function from environments to sets of names:

$$\phi_{\mathcal{E}}(E) : Env \pfun \mathbb{P}\, Name$$

The set of names defined as the free variables for an expression for a particular environment is the smallest set of names which must be in the environment in order for the expression to be well-defined.

However since local declarations do not introduce schema references, the free variables of an expression are unchanged by a local declaration. So in the definitions we omit the environment parameter as it has no effect on the value of the free variables.

Table 22: Free variable functions

Function	Argument
$\phi_{\mathcal{E}}$	Expression
$\phi_{\mathcal{P}}$	Predicate
$\phi_{\mathcal{S}}$	Schema

At the end of each section there is a table defining the free variables for each construct within that category. The following example illustrates the definition of the free variables of a power set:

Table 23: Extract from table of free variables

Expression	Free Variables
...	
Pow s	$\phi_{\mathcal{E}} s$
...	

This can also be read as an equation in the following form:

$$\phi_{\mathcal{E}} \mathbf{Pow}\, s \quad = \quad \phi_{\mathcal{E}} s$$

6.8 Alphabet

The syntactic categories of schema is used to introduce new names. These new names are called the *alphabet*. The alphabet is the set of the names in the signature as defined by the type rules (where applicable).

Table 24: Alphabet function

Function	Argument
α	Schema
α	Schema-text
α	Substitution

Table 25: Extract from table of alphabets

Declaration	Alphabet
$n_1, \ldots, n_m : s$	$\{n_1, \ldots, n_m\}$
...	

This can also be read as an equation in the following form:

$$\alpha(n_1, \ldots, n_m : s) \quad = \quad \{n_1, \ldots, n_m\}$$

6.9 Substitution

The tables of semantic equivalences for substituted expressions are given at the end of each section. These tables indicate when one expression can be replaced by another without changing the meaning. Substitution is defined using a binding, which asssigns values to variable names. These new values are substituted for the variables in the expression.

The following example illustrates the semantic equivalence of substitution into a power set:

Table 26: Extract from table of semantic equivalences

Substitution	Equivalence
...	
$b \circ \mathbb{P}\, u$	$\mathbb{P}(b \circ u)$
...	

This can also be read as an equation in the following form:

$$b \circ \mathbb{P}\, u \quad \equiv \quad \mathbb{P}(b \circ u),$$

where the symbol $\equiv$ denotes semantic equivalence.

Note: The following is an example of substitution:

$$\langle x \rightsquigarrow 5, y \rightsquigarrow \mathbb{N}, a \rightsquigarrow 7 \rangle \circ (x \in y \cap z) \quad \equiv \quad 5 \in \mathbb{N} \cap z$$

Since the variable name a is not free in the expression, there is no substitution for it.

□

7 Expression

Notes on this section of the Z Standard

Section title: Expression
Section editor: This version edited by JEN.
Note : This version reflects comments by Jon Hall, and has been restructured for the proposed concrete syntax.
Contributions by: Stephen Brien, Randolph Johnson, ... *(others to be added)*
Source file: exp.tex
Most recent update: 29th June 1995
Formatted: 3rd July 1995

7.1 Introduction

Expression is a general form for defining values in Z.

In the abstract syntax given below, the different kinds of Z expression are listed.

Abstract Syntax

EXP =	IDENT	Identifier
|	GENINST	Generic Instantiation
|	NUMBERL	Number Literal
|	STRINGL	String Literal
|	SETEXTN	Set Extension
|	SETCOMP	Set Comprehension
|	POWERSET	Power Set
|	TUPLE	Tuple
|	PRODUCT	Cartesian Product
|	TUPLESELECTION	Tuple Selection
|	BINDINGEXTN	Binding Extension
|	THETAEXP	Theta Expression
|	SCHEMAEXP	Schema Expression
|	BINDSELECTION	Binding Selection
|	FUNCTAPP	Application
|	DEFNDESCR	Definite Description
|	IFTHENELSE	Conditional Expression
|	EXPSUBSTITUTION	Substitution

7.2 Method of definition

In this section, definitions are built up in stages: first a *type function* is defined, then a *value function.* From these, a *meaning function* can be derived according to rules given below.

Type function. For any expression $\boldsymbol{E}$, its *type function* $[\![\boldsymbol{E}]\!]^{\mathcal{T}}$ is a recursively defined partial function from type-environments to types:

$$[\![\boldsymbol{E}]\!]^{\mathcal{T}} : \mathit{Tenv} \pfun \mathit{Type}$$

The expression $\boldsymbol{E}$ is *well-typed* in exactly those type-environments contained in $\mathrm{dom}\, [\![\boldsymbol{E}]\!]^{\mathcal{T}}$. The type of an expression in a type-environment is the result of applying its type function to that type-environment.

Value function. For any expression $\boldsymbol{E}$, its *value function* $[\![\boldsymbol{E}]\!]^{\mathcal{V}}$ is a partial function from environments to values:

$$[\![\boldsymbol{E}]\!]^{\mathcal{V}} : \mathit{Env} \pfun W$$

The expression $\boldsymbol{E}$ is *value-defined* in exactly those environments contained in $\mathrm{dom}\, [\![\boldsymbol{E}]\!]^{\mathcal{V}}$.

The value of an expression in an environment is the result of the application of its value function to that environment.

For some expressions the semantics is loosely defined. That is to say, a lower bound on the meaning is given. This provides the possibility of various interpretations, in circumstances where the semantics does not explicitly give a meaning.

> **Note:** An example is the definition of Application. For example, in function application, when the argument is outside the domain of the function, then no meaning is explicitly given. Different interpretations of Z can ascribe different meanings to an ill-formed function application.

Meaning function. The *meaning function* $[\![\boldsymbol{E}]\!]^{\mathcal{M}}$ is a function from environments to elements:

$$[\![\boldsymbol{E}]\!]^{\mathcal{M}} : \mathit{Env} \pfun \mathit{Elm}$$

The meaning of an expression in an environment is the pair consisting of its type and its value in that environment. The type of an expression in an environment is its type evaluated in the corresponding type-environment, which is a restriction of the environment.

The function $\Upsilon \fatsemi [\![\boldsymbol{E}]\!]^{\mathcal{T}}$ corresponds to the type function for $\boldsymbol{E}$ in the full meaning environment, where Υ is the function that restricts an environment to its corresponding type-environment. Thus the meaning function is constructed as follows:

$$[\![\boldsymbol{E}]\!]^{\mathcal{M}} = \langle \Upsilon \fatsemi [\![\boldsymbol{E}]\!]^{\mathcal{T}} , [\![\boldsymbol{E}]\!]^{\mathcal{V}} \rangle$$

The expression $\boldsymbol{E}$ is *well-defined* in exactly those environments contained in $\mathrm{dom}\,[\![\, \boldsymbol{E} \,]\!]^{\mathcal{M}}$ which is equal to:

$$\mathrm{dom}\,\Upsilon \fatsemi [\![\, \boldsymbol{E} \,]\!]^{\mathcal{T}} \cap \mathrm{dom}\,[\![\, \boldsymbol{E} \,]\!]^{\mathcal{V}}$$

An expression is said to be well-typed in an environment if it is well-typed in the corresponding type environment. Thus, an expression is well-defined in those environments in which it is well-typed and value-defined.

A result of this definition is that the type of the meaning of an expression in an environment is always the same as the type part of the expression when evaluated in the corresponding type-environment:

$$\vdash [\![\, \boldsymbol{E} \,]\!]^{\mathcal{M}} \fatsemi t \subseteq \Upsilon \fatsemi [\![\, \boldsymbol{E} \,]\!]^{\mathcal{T}}$$

Note: This relationship is not an equality because it is possible to have well-typed expressions which are not value-defined.

7.3 Identifier

An identifier is a name used to refer to a variable or a constant, and it denotes a value which depends on its environment.

Abstract syntax

IDENT = NAME

Note: A NAME is composed of a *base-name* suffixed by any number of *decorations*.

Concrete form

x

Sample representation and transformation

Representation	Abstract
n	$[\![n]\!]^{\mathcal{E}}$

Type The type of an identifier is the type to which the identifier is mapped in the type-environment:

$$[\![n]\!]^{\mathcal{T}} = (\_n) \rhd Type$$

Note: An identifier is well-typed if and only if it is in the domain of the type-environment.

Value The value of an identifier is the value part of the element mapped to the identifier in the environment:

$$[\![n]\!]^{\mathcal{V}} = (\_n) \fatsemi v$$

Note: An identifier is value-defined in an environment if and only if it is in the domain of the environment.

7.4 Generic instantiation

The generic instantiation $n_{[s_1,\ldots,s_m]}$ $(m > 0)$, is the instantiation of the generic variable n with the list of set expressions $s_1, \ldots, s_m$. The number m must be equal to the number of formal parameters of n. Each element of the instantiation list gives a value to the corresponding generic parameter of the generic variable.

If the list of generic parameters is omitted in the representation form, it is inferred from the typing information in the context of use; in this case, the values of the implicit parameters are the maximal sets of the appropriate type, which must be uniquely determined by the typing rules.

Abstract syntax A generic instantiation is constructed from a variable name and a list of one or more expressions.

GENINST = NAME [EXP, EXP, ..., EXP]

Concrete form

NAME SQBRA ExpressionList1 SQKET

Expression1 , InGen , Expression

PreGen , Expression5

Sample representation and transformation Generic variables can be instantiated by providing a parameter list, or by infix or prefix means.

Representation	Abstract
$n_{[s_1,\ldots,s_m]}$	$[\![n]\!]^{\mathcal{E}}\ [[\![s_1]\!]^{\mathcal{E}}, \ldots, [\![s_m]\!]^{\mathcal{E}}]$
$s_1 \psi s_2$	$[\![(\_\psi\_)]\!]^{\mathcal{E}}\ [[\![s_1]\!]^{\mathcal{E}}, [\![s_2]\!]^{\mathcal{E}}]$
ϕs	$[\![(\phi\_)]\!]^{\mathcal{E}}\ [[\![s]\!]^{\mathcal{E}}]$

Note: The expression $s_1 \psi s_2$, where ψ is an infix generic symbol is the variable $(\_\psi\_)$ when instantiated with the parameter list $[s_1, s_2]$. When ϕ is a prefix generic symbol then ϕs is the variable declared as $(\phi\_)$ when instantiated with the parameter list $[s]$.

Type The type of a generic instantiation $n_{[s_1,\ldots,s_m]}$ is obtained by applying to the types of the actual parameters $s_1, \ldots, s_m$ the function corresponding to the generic type of the variable name n in the type-environment:

$$[\![\ n_{[s_1,\ldots,s_m]}\ ]\!]^{\mathcal{T}} \quad = \quad (\_n) \bullet \langle [\![\ s_1\ ]\!]^{\mathcal{T}}, \ldots, [\![\ s_m\ ]\!]^{\mathcal{T}} \rangle$$

Note: A generic instantiation is well-typed if and only if the variable name is in the domain of the type environment and there is a correct number of set-typed parameters.

Value The value of a generic instantiation $n_{[s_1,\ldots s_m]}$ is obtained by applying the function corresponding to the generic meaning of the variable name n in the environment to the meanings of the actual parameters $s_1, \ldots s_m$ and then taking the value part:

$$[\![n_{[s_1,\ldots,s_m]}]\!]^{\nu} = ((\_n) \bullet \langle [\![s_1]\!]^{\mathcal{M}}, \ldots, [\![s_m]\!]^{\mathcal{M}} \rangle) \fatsemi v$$

Note: A generic instantiation is value-defined if and only if it is well-typed and all its parameters are value-defined.

7.5 Number literal

A number literal denotes an integer.

Abstract syntax

NUMBERL = NUMBER

Concrete form

NUMBER

Sample representation and transformation

Representation	Abstract
i	$[\![i]\!]^{\mathcal{E}}$

Type The type of a number literal is the given set type of the integers.

$$[\![i]\!]^{\mathcal{T}} = \mathbb{Z}^{\circ} ; givenT$$

Note: A number literal is always well-typed.

Value The value of a number literal is the integer it denotes.

$$[\![i]\!]^{\mathcal{V}} = i^{\circ}$$

Note: A number literal is always value-defined.

7.6 String literal

A string literal denotes a string.

Abstract syntax

STRINGL = STRING

Concrete form

STRING

Sample representation and transformation

Representation	Abstract
a	$(\!\lvert a \rvert\!)^{\mathcal{E}}$

Type The type of a string literal is the given set type of the set $\mathbb{S}$ of strings.

$$[\![\, a \,]\!]^{\mathcal{T}} = \mathbb{S}^{\circ} \fatsemi \mathit{givenT}$$

Note: A string literal is always well-typed.

Value The value of a string literal is the string it denotes.

$$[\![\, a \,]\!]^{\mathcal{V}} = a^{\circ}$$

Note: A string literal is always value-defined.

7.7 Set extension

A non-empty set extension $\{x_1, \ldots, x_m\}$ is a set containing exactly those elements denoted by $x_1, \ldots, x_m$ $(m > 0)$.

Note: Since a set is characterised by its members, the order and duplication of elements in $x_1, \ldots, x_m$ is of no consequence.

Abstract syntax A set extension is constructed from a list of one or more expressions.

$$\mathsf{SETEXTN} = \{\mathsf{EXP}, \mathsf{EXP}, \ldots, \mathsf{EXP}\}$$

Concrete form

SETBRA ExpressionList SETKET

'⟨' , Expression0 , {',' , Expression0} , '⟩'

'⟦' , Expression0 , {',' , Expression0} , '⟧'

Sample representation and transformation There are three kinds of sets which can be constructed by extension: simple sets, sequences, and bags.

Representation	Abstract
$\{x_1, \ldots, x_m\}$	$\{[\![x_1]\!]^{\mathcal{E}}, \ldots, [\![x_m]\!]^{\mathcal{E}}\}$
$\langle x_1, \ldots, x_m \rangle$	$[\![\{(1, x_1), \ldots, (m, x_m)\}]\!]^{\mathcal{E}}$
$[\![x_1, \ldots, x_m]\!]$	$[\![\{(x_1, 1)\} \uplus \ldots \uplus \{(x_m, 1)\}]\!]^{\mathcal{E}}$

Note: The expression $\langle x_1, \ldots, x_m \rangle, (m > 0)$ defines an explicit construction of a sequence, which can be regarded as an ordered collection of its constituents. A sequence is modelled as a partial function mapping the numbers $1, \ldots, m$ to the expressions $x_1, \ldots, x_m$ respectively.

Note: The expression $[\![x_1, \ldots, x_m]\!], (m > 0)$ defines an explicit construction of a bag. A bag is a collection of possibly multiply-occurring elements. A bag is modelled as a partial function mapping its constituents to the number of times they occur within the bag.

Type The type of a set extension $\{x_1, \ldots, x_m\}$ is the power set type of the common type of $x_1, \ldots, x_m$.

$$[\![\{x_1, \ldots, x_m\}]\!]^{\mathcal{T}} = ([\![x_1]\!]^{\mathcal{T}} \cap \ldots \cap [\![x_m]\!]^{\mathcal{T}}) \, ; \, powerT$$

Note: A set extension $\{x_1, \ldots, x_m\}$ is well typed if and only if all of the expressions $x_1, \ldots, x_m$ are well-typed with the same type.

Note: If σ represents the common type of $x_1, \ldots, x_m$, then $\mathbb{P}\,\sigma$ represents the type of the set extension $\{x_1, \ldots, x_m\}$, $\mathbb{P}(\mathbb{Z} \times \sigma)$ represents the type of the sequence $\langle x_1, \ldots, x_m \rangle$ and $\mathbb{P}(\sigma \times \mathbb{Z})$ represents the type of the bag $[\![x_1, \ldots, x_m]\!]$.

Value The value of a set extension $\{x_1, \ldots, x_m\}$ is the set of the values of $x_1, \ldots, x_m$:

$$[\![\, \{x_1, \ldots, x_m\} \,]\!]^{\nu} \;=\; \langle [\![\, x_1 \,]\!]^{\nu}, \ldots, [\![\, x_m \,]\!]^{\nu} \rangle \,;\, \{..\}$$

Note: A set extension $\{x_1, \ldots, x_m\}$ is value-defined if and only if all of $x_1, \ldots, x_m$ are value-defined.

Note: Two sets $\{x_1, \ldots, x_m\}$ and $\{\, y_1, y_2, \ldots, y_m \,\}$ are equal if and only if for all i there exists j such that $x_i = y_j$, $1 \leq i \leq n$
and for all j there exists k such that $y_j = x_k$, $1 \leq j \leq m$

7.8 Set comprehension

The set comprehension $\{St \bullet x\}$ is the set that contains exactly those elements denoted by the expression x when evaluated in each enrichment of the current environment by the schema text St.

Abstract syntax A set comprehension is constructed from a schema text and an expression.

SETCOMP = {SCHEMA • EXP}

Concrete form

SETBRA TextOrExpression DOT Expression SETKET

'{' , SchemaText, '}'

'λ' , SchemaText, '•' , Expression

Sample representation and transformation There are two ways of constructing a set by comprehension: a simple set (for which the expression part is optional) and a lambda expression.

Representation	Abstract
$\{St \bullet x\}$	$\{[\![St]\!]^{\mathcal{ST}} \bullet [\![x]\!]^{\mathcal{E}}\}$
$\{St\}$	$\{[\![St]\!]^{\mathcal{ST}} \bullet [\![(St)^{\chi}]\!]^{\mathcal{E}}\}$
$\lambda St \bullet x$	$\{[\![St]\!]^{\mathcal{ST}} \bullet ([\![(St)^{\chi}]\!]^{\mathcal{E}}, [\![x]\!]^{\mathcal{E}})\}$

Note: If the expression part of the set comprehension is omitted then the default is the characteristic tuple of the schema text.

Note: A lambda expression denotes a function. The parameter is the characteristic tuple of the SchemaText. The domain is defined by the SchemaText. The value of the function for a given parameter is defined by the value of the Expression for the value of that parameter.

Editor's note: A definition of χ is needed here.

Type The type of a set comprehension $\{St \bullet x\}$ is the power set type of the type of x in the type-environment enriched by the declaration St:

$$[\![\{St \bullet x\}]\!]^{\tau} = (St)^{\tau} ; [\![x]\!]^{\tau} ; powerT$$

Note: A set comprehension $\{St \bullet x\}$ is well-typed if and only if St is well-typed, and x is well-typed in the type-environment enriched by St.

Value The value of a set comprehension $\{St \bullet x\}$, is the set of the values denoted by the expression x in each of the enrichments of the environment by the schema text St:

$$[\![\{St \bullet x\}]\!]^{\nu} = {}^{\Lambda}(\langle St \rangle^{\mathcal{M}} ; [\![x]\!]^{\nu})$$

Note: A set comprehension is always value-defined.

7.9 Power set

The power set $\mathbb{P}\, s$ is the set of all subsets of the set s.

Abstract syntax A power set is constructed from an expression.

POWERSET = **Pow** EXP

Concrete form

`PSET Expression`

Sample representation and transformation

Representation	Abstract
$\mathbb{P}\, s$	$\mathbf{Pow}[\![s]\!]^{\mathcal{E}}$

Type The type of the power set $\mathbb{P}\, s$ is the power set type of the type of the set s.

$$[\![\, \mathbf{Pow}\, s \,]\!]^{\mathcal{T}} \;=\; ([\![\, s \,]\!]^{\mathcal{T}} \rhd \mathit{Ptype}) \fatsemi \mathit{powerT}$$

Note: A power set $\mathbb{P}\, s$ is well-typed if and only if s has power set type.

Note: If $\mathbb{P}\, \sigma$ represents the type of the set s, then $\mathbb{P}\, \mathbb{P}\, \sigma$ represents the type of $\mathbb{P}\, s$, a set of sets.

Value The value of the power set $\mathbb{P}\, s$ is the set of all the subsets of the value of s:

$$[\![\, \mathbf{Pow}\, s \,]\!]^{\mathcal{V}} \;=\; [\![\, s \,]\!]^{\mathcal{V}} \fatsemi (\mathbb{P})$$

Note: A power set $\mathbb{P}\, s$ is value-defined if and only if the expression s is value-defined.

7.10 Tuple

A tuple $(x_1, \ldots, x_m)$ $(m > 1)$ is an ordered collection of the elements $x_1, \ldots, x_m$. The elements $x_1, \ldots, x_m$ are not required to have the same type.

Note: The tuples (a, b, c) and $((a, b), c)$ are distinct: the first contains three components a, b, c whereas the second has components (a, b) and c.

Note: The expression (a) is not a tuple; it is the expression a within parentheses.

Abstract syntax A tuple is constructed from a list of two or more expressions.

$\mathsf{TUPLE} = (\mathsf{EXP}, \mathsf{EXP}, \ldots, \mathsf{EXP})$

Concrete form

`BRA Expression COMMA ExpressionList1 KET`

Sample representation and transformation

Representation	Abstract
$(x_1, \ldots, x_m)$	$(\llbracket x_1 \rrbracket^{\mathcal{E}}, \ldots, \llbracket x_m \rrbracket^{\mathcal{E}})$

Type The type of a tuple $(x_1, \ldots, x_m)$ is the Cartesian product type formed from the types of $x_1, \ldots, x_m$:

$$\llbracket (x_1, \ldots, x_m) \rrbracket^{\mathcal{T}} = \langle \llbracket x_1 \rrbracket^{\mathcal{T}}, \ldots, \llbracket x_m \rrbracket^{\mathcal{T}} \rangle \fatsemi cproductT$$

Note: A tuple $(x_1, \ldots, x_m)(m > 1)$ is well-typed if and only if all of $x_1, \ldots, x_m$ are well-typed.

Value The value of a tuple $(x_1, \ldots, x_m)(m > 1)$ is the tuple formed from the values of $x_1, \ldots, x_m$:

$$\llbracket (x_1, \ldots, x_m) \rrbracket^{\mathcal{V}} = \langle \llbracket x_1 \rrbracket^{\mathcal{V}}, \ldots, \llbracket x_m \rrbracket^{\mathcal{V}} \rangle$$

Note: A tuple $(x_1, \ldots, x_m)$ is value-defined if and only if all of $x_1, \ldots, x_m$ are value-defined.

7.11 Cartesian product

The expression $s_1 \times \ldots \times s_m$ $(m > 1)$ is the Cartesian product of the sets $s_1, \ldots, s_m$.

The sets $s_1, \ldots, s_m$ are not required to have the same type.

Note: As with tuples, the Cartesian products $a \times b \times c$ and $(a \times b) \times c$ are distinct.

Abstract syntax A Cartesian product is constructed from two or more expressions.

$$\mathsf{PRODUCT} = \mathsf{EXP} \times \mathsf{EXP} \times \ldots \times \mathsf{EXP}$$

Concrete form

```
Expression CROSS Expression CROSS Expression
```

Sample representation and transformation

Representation	Abstract
$s_1 \times \ldots \times s_m$	$[\![s_1]\!]^{\mathcal{E}} \times \ldots \times [\![s_m]\!]^{\mathcal{E}}$

Type The type of a Cartesian product $s_1 \times \ldots \times s_m (m > 1)$ is the power set type of the Cartesian product type of the list of the underlying types of the sets $s_1, \ldots, s_m$.

$$[\![s_1 \times \ldots \times s_m]\!]^{\mathcal{T}} = \langle [\![s_1]\!]^{\mathcal{T}} \fatsemi powerT^{-1}, \ldots, [\![s_m]\!]^{\mathcal{T}} \fatsemi powerT^{-1} \rangle \fatsemi cproductT \fatsemi powerT$$

Note: A Cartesian product $s_1 \times \ldots \times s_m$ is well-typed if and only if all of the elements $(s_1, \ldots, s_m)$ have power set types.

Value The value of a Cartesian product $s_1 \times \ldots \times s_m (m > 1)$ is the Cartesian product of the values of the sets $(s_1, \ldots, s_m)$:

$$[\![s_1 \times \ldots \times s_m]\!]^{\mathcal{V}} = \langle [\![s_1]\!]^{\mathcal{V}}, \ldots, [\![s_m]\!]^{\mathcal{V}} \rangle \fatsemi \times$$

Note: A Cartesian product $s_1 \times \ldots \times s_n$ is value-defined if and only if all of the sets $s_1, \ldots, s_n$ are value-defined.

Note: If $x_i \in s_i$ for $1 \leq i \leq m$, then the tuple $(x_1, \ldots, x_m)$ is an element of $s_1 \times \ldots \times s_m$.

7.12 Tuple selection

The tuple selection $t.i$ is the ith element in the tuple t.

Abstract syntax A tuple selection is constructed from an expression and a number literal.

TUPLESELECTION = EXP . NUMBERL

Note: The syntactic category NUMBERL is used to ensure well-typedness of selection.

Concrete form

`Expression SELECT NUMBER`

Sample representation and transformation

Representation	Abstract
$t.i$	$[\![t]\!]^{\mathcal{E}}.[\![i]\!]^{\mathcal{E}}$

Type The type of a tuple selection $t.i$ is the type of the ith element of the tuple t.

$$[\![\, t.i \,]\!]^{\mathcal{T}} \;=\; [\![\, t \,]\!]^{\mathcal{T}} \fatsemi cproductT^{-1} \fatsemi \pi_i$$

Note: The tuple selection $t.i$ is well-typed if and only if t has a Cartesian product type with at least i elements.

Value The value of a tuple selection $t.i$ is the value of the ith element of the tuple t.

$$[\![\, t.i \,]\!]^{\mathcal{V}} \;=\; [\![\, t \,]\!]^{\mathcal{V}} \fatsemi \pi_i$$

Note: The tuple selection $t.i$ is value-defined if and only if t has the value of a tuple with at least i elements.

7.13 Binding extension

A binding extension $\langle\!| n_1 \rightsquigarrow x_1, \ldots, n_m \rightsquigarrow x_m |\!\rangle$ $(m > 0)$ is the binding that maps the names $n_1, \ldots, n_m$ to the values of the expressions $x_1, \ldots, x_m$ respectively.

Abstract syntax A binding extension is constructed from a list of names and expressions.

$$\mathsf{BINDINGEXTN} = \langle\!| \mathsf{NAME} := \mathsf{EXP}, \ldots, \mathsf{NAME} := \mathsf{EXP} |\!\rangle$$

Concrete form

`BINDERBRA BindList BINDERKET`

Sample representation and transformation

Representation	Abstract
$\langle\!\| n_1 \rightsquigarrow x_1, \ldots, n_m \rightsquigarrow x_m \|\!\rangle$	$\langle\!\| [\![n_1]\!]^{\mathcal{E}} \rightsquigarrow [\![x_1]\!]^{\mathcal{E}}, \ldots, [\![n_m]\!]^{\mathcal{E}} \rightsquigarrow [\![x_m]\!]^{\mathcal{E}} \|\!\rangle$

Type The type of a binding extension $\langle\!| n_1 \rightsquigarrow x_1, \ldots, n_m \rightsquigarrow x_m |\!\rangle$ $(m > 0)$ is the schema type of the signature constructed from the mapping of the names $n_1, \ldots, n_m$ to the types of the expressions $x_1, \ldots, x_m$.

$$[\![\langle\!| n_1 \rightsquigarrow x_1, \ldots, n_m \rightsquigarrow x_m |\!\rangle]\!]^{\mathcal{T}} = \langle\langle n_1^\circ, [\![x_1]\!]^{\mathcal{T}}\rangle, \ldots, \langle n_m^\circ, [\![x_m]\!]^{\mathcal{T}}\rangle\rangle \fatsemi \{..\} \fatsemi schemaT$$

Note: A binding extension $\langle\!| n_1 \rightsquigarrow x_1, \ldots, n_m \rightsquigarrow x_m |\!\rangle$ is well-typed if and only if the expressions $x_1, \ldots, x_m$ are all well-typed, and the names are distinct.

Value The value of a binding extension $\langle\!| n_1 \rightsquigarrow x_1, \ldots, n_m \rightsquigarrow x_m |\!\rangle$ $(m > 0)$ is the binding constructed from the mapping of the names $n_1, \ldots, n_m$ to the values of the expressions $x_1, \ldots, x_m$.

$$[\![\langle\!| n_1 \rightsquigarrow x_1, \ldots, n_m \rightsquigarrow x_m |\!\rangle]\!]^{\mathcal{V}} = \langle\langle n_1^\circ, [\![x_1]\!]^{\mathcal{V}}\rangle, \ldots, \langle n_m^\circ, [\![x_m]\!]^{\mathcal{V}}\rangle\rangle \fatsemi \{..\}$$

Note: A binding extension $\langle\!| n_1 \rightsquigarrow x_1, \ldots, n_m \rightsquigarrow x_m |\!\rangle$ is value-defined if and only if the expressions $x_1, \ldots, x_m$ are all value-defined.

7.14 Theta expression

The theta expression $\theta\, S$ is the binding whose type is constructed from the signature of S and whose value is the binding constructed from the mapping of the names of the signature to their values in the environment. The theta expression $\theta\, S\,^{q}$ is the binding whose type is constructed from the signature of S and whose value is the binding constructed from the mapping of the names of the signature to the values in the environment of those names when decorated by ^{q}.

Abstract syntax A theta expression is constructed from a schema and an optional decoration.

$$\mathsf{THETAEXP} \;=\; \theta\ \mathsf{SCHEMA}\ \mathsf{DECOR}$$

Note: The schema may itself be decorated. Thus the following are permitted: $\theta\, S\,^{q}$ and $\theta\,(S^{q})\,^{q}$.

Note: Only non-generic schemas may be used in theta expressions.

Concrete form

```
THETA Expression
```

Sample representation and transformation

Representation	Abstract
$\theta\ S^{q}$	$\theta[\![S]\!]^{\mathcal{S}}\,[\![q]\!]^{\mathcal{E}}$
$\theta\ S$	$\theta[\![S]\!]^{\mathcal{S}}$

Type The type of $\theta\, S$ ($\theta\, S\,^{q}$) is the schema type constructed from the signature of S whose components (when decorated by ^{q}) have the same non-generic type as the corresponding variable in the type-environment:

$$[\![\, \theta S \,]\!]^{\mathcal{T}} \;=\; (\!(\, S \,)\!)^{\mathcal{T}} \cap \supseteq) \,;\, schemaT$$
$$[\![\, \theta S^{q} \,]\!]^{\mathcal{T}} \;=\; \Big((\!(\, S \,)\!)^{\mathcal{T}} \cap (\sqsupseteq;\, ^{\exists}(\!(\, q \,)\!)^{\mathcal{N}} \;\otimes\; 1))\Big) \,;\, schemaT$$

Note: A theta expression is well-typed if and only if each of the decorated versions of the names of the signature of the schema is assigned a non-generic type in the type-environment which is the same as the type of that name in the signature.

Note: The type of a theta expression $\theta\, S^{q}$ is *not* the type taken from S decorated by ^{q}. The decoration ^{q} does *not* necessarily appear in the resulting type. The use of the schema is to identify the type of the resulting binding. Decoration is used only to identify which names to look up in the type-environment; thus $\theta\, S^{r}$ and $\theta\, S^{q}$ are of the same type even if ^{r} and ^{q} are different decorations.

Value The value of the theta expression $\theta\, S$ ($\theta\, S^{q}$) is a binding of the names of the components of S to the values of the names (when decorated by ^{q}) in the environment:

$$[\![\, \theta S \,]\!]^{\nu} = \Upsilon \fatsemi (\!|\, S \,|\!)^{\tau} \fatsemi schemaT \fatsemi Elm \cap \supseteq \fatsemi V$$
$$[\![\, \theta S^{q} \,]\!]^{\nu} = \Upsilon \fatsemi (\!|\, S \,|\!)^{\tau} \fatsemi schemaT \fatsemi Elm \cap \supseteq \fatsemi {}^{\exists}((\!|\, q \,|\!)^{\mathcal{N}} \otimes v)$$

Note: A well-typed theta expression is always value-defined. The value of the theta-expression does not have to satisfy the property of the schema.

7.15 Schema expression

A schema expression S is the set of bindings defined by the schema S.

Abstract syntax A schema expression is constructed from a schema.

SCHEMAEXP = SCHEMA

Concrete form

SCHEMA (NB add to syntax)

Sample representation and transformation

Representation	Abstract
S	$[\![S]\!]^S$

Type The type of a schema expression S is the power set type of the schema type constructed from the signature of the schema S:

$$[\![S]\!]^T = (\!| S |\!)^T ; schemaT ; powerT$$

Note: A schema expression S is well-typed if and only if the schema S is well-typed.

Note: The type of a schema expression is not in the range of $schemaT$: it is in the range of $schemaT ; powerT$. The relationship between $(\!| \ |\!)^T$ and $[\![\ ]\!]^T$ is that of $schemaT ; powerT$.

Value The value of a schema expression S is the set of bindings defined by the schema S:

$$[\![S]\!]^V = {}^\wedge((\!| S |\!)^M ; V)$$

Note: A schema expression S is always value-defined.

7.16 Binding selection

The binding selection $\boldsymbol{b.n}$ is the element to which the name $\boldsymbol{n}$ is mapped in the binding $\boldsymbol{b}$.

Abstract syntax A binding selection is constructed from a binding and a name.

BINDSELECTION = EXP . NAME

Concrete form

`Expression SELECT NAME`

Sample representation and transformation

Representation	Abstract
$b.n$	$[\![b]\!]^{\mathcal{E}}.[\![n]\!]^{\mathcal{E}}$

Type The type of a binding selection $\boldsymbol{b.n}$ is the type to which the name $\boldsymbol{n}$ is mapped in the signature used to construct the schema type of the binding $\boldsymbol{b}$:

$$[\![b.\, n]\!]^{\mathcal{T}} \;=\; [\![b]\!]^{\mathcal{T}} \fatsemi schemaT^{-1} \fatsemi (\_n)$$

Note: A binding selection $\boldsymbol{b.n}$ is well-typed if and only if the type of $\boldsymbol{b}$ is a schema type and the name $\boldsymbol{n}$ is in the domain of the signature from which the schema type is constructed.

Value The value of a binding selection $\boldsymbol{b.n}$ is the value to which the name $\boldsymbol{n}$ is mapped in the binding $\boldsymbol{b}$:

$$[\![b.\, n]\!]^{\mathcal{V}} \;=\; [\![b]\!]^{\mathcal{V}} \fatsemi (\_n)$$

Note: A binding selection $\boldsymbol{b.n}$ is value-defined if and only if the binding $\boldsymbol{b}$ is value-defined and the name $\boldsymbol{n}$ is in its domain.

Note: Two bindings x and y with components $n_1, \ldots, n_m$ are equal if and only if $x.n_i = y.n_i$, $1 \leq i \leq m$.

7.17 Application

The application $f\ x$ is the result of applying f to the argument x.

Abstract syntax

FUNCTAPP = EXP (EXP)

Sample representation and transformation There are four ways of representing an application: a prefix form, an infix form, a superscript form and a postfix form. For applications declared for use in postfix or infix form, underscores indicate the positions of the operands. The complete name includes the underscores and surrounding parentheses which are omitted when the operands are supplied in the form defined in the declaration.

Concrete form

```
Prefix Expression
xx
xxx
xxxx
```

Representation	Abstract
$f\ x$	$[\![f]\!]^{\mathcal{E}}\ [\![x]\!]^{\mathcal{E}}$
$x \phi y$	$(\_ \phi \_)[\![(x, y)]\!]^{\mathcal{E}}$
R^x	$(iter [\![x]\!]^{\mathcal{E}})[\![R]\!]^{\mathcal{E}}$
$x \phi$	$(\_ \phi)[\![x]\!]^{\mathcal{E}}$

Note: The application $x\ \phi\ y$ is the infix application of the relation $(\_ \phi \_)$ applied to the pair of arguments (x, y).

Note: The application R^x denotes the x-iteration of the relation R; it is an abbreviation of the expression $iter\ x\ R$.

Note: The application $x\phi$ is the postfix application of the relation $(\_ \phi)$ applied to the argument x.

Type In the expression $f\ x$ the type of f must be the power set type of the Cartesian product type of a pair of types, and the type of the argument x must be the first type in this pair; the type of $f\ x$ is the second type in the pair.

$$[\![f\ x]\!]^{\tau} = \left([\![f]\!]^{\tau} \fatsemi power T^{-1} \fatsemi cproduct T^{-1} \fatsemi \{-\}\right) \bullet [\![x]\!]^{\tau}$$

Note: The application $f\ x$ is well-typed only if the type of f is a power set type of a pair of types with the first type in the pair the same as the type of x.

Note: If we evaluate the type of f, we get essentially a set of pairs, where each pair comprises the type of an argument and the type of its result. If we next evaluate the type of the particular argument x, then we can simply use the type of f as a function to look up the type of the result corresponding to x. We say the the type of f is essentially a set of pairs, because we must 'undo' the type constructors.

Value The value of an application $f\ x$ is given by applying the value of f to the value of the argument x:

$$[\![f\ x]\!]^{\nu} \supseteq {}^{\wedge}([\![f]\!]^{\nu} \bullet [\![x]\!]^{\nu}) \fatsemi \{-\}^{-1}$$

Note: A well-typed application $f\ x$ is value-defined if both f and x are value-defined and if there is a unique w such that $(x, w) \in f$.

Note: A relation is a set of pairs; the first element of each pair represents an argument, and the second the result for that argument. For the application $f\ x$ to be defined, f must be functional at x. Providing that x evaluates in the environment ρ to a value v, and the value of f in ρ contains (v, w), and no other pair starting with v, then the expression $(f\ x)$ evaluates to w. So for a well-defined function application we would expect an equality of the following form:

$$[\![f\ x]\!]^{\nu}_{\rho} = [\![f]\!]^{\nu}_{\rho}\ ([\![x]\!]^{\nu}_{\rho})$$

The promoted application $[\![f]\!]^{\nu} \bullet [\![x]\!]^{\nu}$ provides a satisfactory meaning when the application is value-defined. It is necessary to decide what to do with $f\ x$ when f is not functional at x. This arises if there are several pairs in the value of f, each having the same first element equal to the value of x or if there is none. The definition provided does not prescribe a value for a relation applied outside its domain or where it is non-functional.

7.18 Definite description

The definite description $\mu\, St \bullet x$ is the element denoted by x in the unique enrichment of the environment by the schema text St.

Abstract syntax A definite description is constructed from a schema text and an expression.

$$\mathsf{DEFNDESCR} = \mu\, \mathsf{SCHEMA} \bullet \mathsf{EXP}$$

Concrete form

```
BRA MU TextOrExpression KET
```

Sample representation and transformation In the representation form for definite description, the expression part is optional.

Representation	Abstract
$\mu\, St \bullet x$	$\mu [\![St]\!]^{\mathcal{ST}} \bullet [\![x]\!]^{\mathcal{E}}$
$\mu\, St$	$\mu [\![St]\!]^{\mathcal{ST}} \bullet [\![(St)^{\chi}]\!]^{\mathcal{E}}$

Note: If the expression part of the definite description is omitted then the default is the characteristic tuple of the schema text.

Type The type of the term $\mu\, St \bullet x$ is the type of x in the type-environment enriched by St:

$$[\![\, \mu\, St \bullet x \,]\!]^{\mathcal{T}} = (\!|\, St \,|\!)^{\mathcal{T}} \fatsemi [\![\, x \,]\!]^{\mathcal{T}}$$

Note: The expression $\mu\, St \bullet x$ is well-typed if and only if St is well-typed, and x is well-typed in the type-environment enriched by St.

Value The value of a definite description $\mu\, St \bullet x$ is the value of x in the unique enrichment of the environment by St:

$$[\![\, \mu\, St \bullet x \,]\!]^{\mathcal{V}} \supseteq {}^{\wedge}((\!|\, St \,|\!)^{\mathcal{M}}) \fatsemi \{-\}^{-1} \fatsemi [\![\, x \,]\!]^{\mathcal{V}}$$

Note: A well-typed definite description $\mu\, St \bullet x$ is value-defined if there is exactly one defined enrichment of the environment by the schema text St and the expression x is value-defined in that enriched environment.

Note: This definition is not specific about the value of an improper definite description. If there is no unique enrichment of the environment then the value is not prescribed; hence the use of $\supseteq$ in the definition.

7.19 Conditional expression

The conditional expression **if** P **then** x **else** y denotes an expression which is equal to x if the predicate P is true, otherwise it is equal to the expression y.

Abstract syntax A conditional expression is constructed from a predicate and two expressions.

IFTHENELSE = **if** PRED **then** EXP **else** EXP **fi**

Concrete form

```
IF Predicate THEN Expression ELSE Expression
```

Sample representation and transformation

Representation	Abstract
If P Then x Else y	$\mathbf{if}(\!\lvert P \rvert\!)^{\mathcal{P}} \mathbf{then} (\!\lvert x \rvert\!)^{\mathcal{E}} \mathbf{else} (\!\lvert y \rvert\!)^{\mathcal{E}}$

Type The type of the conditional expression **if** P **then** x **else** y is the common type of the expressions x and y when the predicate P is well-typed:

$$[\![\mathbf{if}\ P\ \mathbf{then}\ x\ \mathbf{else}\ y]\!]^{\mathcal{T}} = \{\!|P|\!\}^{\mathcal{T}} \lhd ([\![x]\!]^{\mathcal{T}} \cap [\![y]\!]^{\mathcal{T}})$$

Note: The expression **if** P **then** x **else** y is well-typed if and only if the predicate P is well-typed and the expressions x and y are both well-typed with the same type.

Value The value of the conditional expression **if** P **then** x **else** y is the value of the expression x when the predicate P is true, otherwise it is the value of the expression y:

$$[\![\mathbf{if}\ P\ \mathbf{then}\ x\ \mathbf{else}\ y]\!]^{\mathcal{V}} = (\{\!|P|\!\}^{\mathcal{M}} \lhd [\![x]\!]^{\mathcal{V}}) \cup (\{\!|P|\!\}^{\mathcal{M}} \lhd [\![y]\!]^{\mathcal{V}})$$

Note: The expression **if** P **then** x **else** y is value-defined if and only if the predicate P is true and the expression x is value-defined or the predicate $\neg P$ is true and the expression y is value-defined.

7.20 Substitution

The expression $b \odot x$ denotes an expression equal to x in the environment enriched by the binding b.

Abstract syntax An expression substitution is constructed from a binding and an expression.

$$\mathsf{EXPSUBSTITUTION} = \mathsf{EXP} \odot \mathsf{EXP}$$

Concrete form

`Expression SUBST Expression`

Sample representation and transformation

Representation	Abstract
$b \odot x$	$[\![b]\!]^{\mathcal{E}} \odot [\![x]\!]^{\mathcal{E}}$

Type The type of the substitution $b \odot x$ is the type of the expression x in the type-environment enriched by the binding b.

$$[\![\, b \odot x \,]\!]^{\mathcal{T}} = \langle 1, [\![\, b \,]\!]^{\mathcal{T}} \fatsemi schemaT^{-1} \rangle \fatsemi \oplus \fatsemi [\![\, x \,]\!]^{\mathcal{T}}$$

Note: The substitution $b \odot x$ is well-typed if and only if b has schema-type and the expression x is well-typed in the type-environment enriched by the binding b.

Value The value of the substitution $b \odot x$ is the value of the expression x in the environment enriched by the binding b.

$$[\![\, b \odot x \,]\!]^{\mathcal{V}} = \langle 1, [\![\, b \,]\!]^{\mathcal{M}} \fatsemi \diamond \rangle \fatsemi \oplus \fatsemi [\![\, x \,]\!]^{\mathcal{V}}$$

Note: The substitution $b \odot x$ is value-defined if and only if b is value-defined and the expression x is value-defined in the environment enriched by the binding b.

Editor's note: Revised versions of the following subsections have been included in Annex F, *The logical theory of Z*.

Free variables

Substitution

□

8 Predicate

Notes on this section of the Z Standard

Section title: Predicate
Section editor: this version, edited by JEN
Original text by: Stephen Brien
Contributions by: Stephen Brien, ... *(others to be added)*
Source file: pred.tex
Notes: Updated for new syntax
Most recent update: 29th June 1995
Formatted: 3rd July 1995

8.1 Introduction

A *Predicate* is the general form for expressing properties of the environment. These properties are relationships between the values of the variables in the environment.

In the abstract syntax below the different kinds of predicate are listed.

Abstract syntax

PRED	= EQUALITY	Equality
	| MEMBERSHIP	Set Membership
	| TRUTH	Truth Literal
	| FALSEHOOD	False Literal
	| NEGATION	Negation
	| DISJUNCTION	Disjunction
	| CONJUNCTION	Conjunction
	| IMPLICATION	Implication
	| EQUIVALENCE	Equivalence
	| UNIVERSALQUANT	Universal Quantification
	| EXISTSQUANT	Existential Quantification
	| UNIQUEQUANT	Unique Existential Quantification
	| SPRED	Schema Predicate
	| PREDSUBSTITUTION	Substitution

Strategy for definition

The description of the meaning of a predicate is split into two parts. The first part gives rules for determining whether it is well-typed or not. The second determines whether the predicate is ZF-true in the environment.

A predicate is *ZF-true* in an environment if the values of the sub-expressions in the predicate are such

that the predicate is true in that environment, without necessarily considering whether the predicate is well-typed.

8.1.1 Type

Since in the abstract syntax we already know that a certain construct is a predicate, when considering the type of a predicate the only matter of concern is whether it is well-typed. For this reason we represent the type function of a predicate as the set of type-environments in which it is well-typed.

$$(\!|\text{PRED}|\!)^{\mathcal{T}} : \mathbb{P}\, Tenv$$

Note: The predicate $x = y$ is meaningless if the expressions x and y are not of the same type. There is no meaningful way of comparing them.

Note: A predicate that is not well-typed in any environment has a type function that evaluates to the empty set.

8.1.2 Value

The value function maps a predicate to the set of environments in which it is ZF-true:

$$(\!|\text{PRED}|\!)^{\mathcal{V}} : \mathbb{P}\, Env$$

Note: The predicate $\neg(x \in x)$ is ZF-true in all environments. This is so because, within the semantic universe, the axiom of regularity ensures that $x \in x$ is false and hence $\neg(x \in x)$ is true. On the other hand, in Z, the type-system ensures that $x \in x$ is not well-typed so therefore $\neg(x \in x)$ is not well-typed.

8.1.3 Meaning

The environments in which a predicate is true are exactly those environments in which the predicate is well-typed and is ZF-true.

$$(\!|\text{PRED}|\!)^{\mathcal{M}} : \mathbb{P}\, Env$$

$$(\!|\text{PRED}|\!)^{\mathcal{M}} == {}^{\exists}(\Upsilon^{-1})(\!|\text{PRED}|\!)^{\mathcal{T}} \cap (\!|\text{PRED}|\!)^{\mathcal{V}}$$

Note: As indicated in the note above, the predicate $\neg(x \in x)$ is ZF-true but not well-typed, hence it is not true in any environment. The meaning of the predicate is the empty set: $(\!|x \in x|\!)^{\mathcal{M}} = \varnothing$.

8.2 Equality

Two expressions are equal if and only if they have the same value and type.

Abstract syntax An equality is constructed from two expressions.

EQUALITY = EXP = EXP

Concrete form

`Expression EQUALS Expression`

Note: This form is derived from `Relation` and `InfixRel`

Sample representation and transformation

Representation	Abstract
$[\![x = y]\!]^{\mathcal{P}}$	$[\![x]\!]^{\mathcal{E}}=[\![y]\!]^{\mathcal{E}}$

Type An equality $x = y$ is well-typed in those environments in which the expressions x and y have the same type.

$$\{\!|x = y|\!\}^{\mathcal{T}} = \mathrm{dom}([\![x]\!]^{\mathcal{T}} \cap [\![y]\!]^{\mathcal{T}})$$

Value An equality $x = y$ is ZF-true in those environments in which the expressions x and y have the same values.

$$\{\!|x = y|\!\}^{\mathcal{V}} = \mathrm{dom}([\![x]\!]^{\mathcal{V}} \cap [\![y]\!]^{\mathcal{V}})$$

8.3 Membership

The predicate $x \in y$ is true if and only if the expression x is a member of the set denoted by the expression y.

Abstract syntax A membership predicate is constructed from two expressions.

$$\mathsf{MEMBERSHIP} = \mathsf{EXP} \in \mathsf{EXP}$$

Concrete form

`Expression MEMBER Expression`

Note: This form is derived from `Relation` and `InfixRel`

Sample representation and transformation There are three ways in which the membership predicate can be written: using the membership sign, using an infix relation and using a prefix relation.

Representation	Abstract
$x \in y$	$[\![x]\!]^{\mathcal{E}} \in [\![y]\!]^{\mathcal{E}}$
$x \rho y$	$[\![(x, y)]\!]^{\mathcal{E}} \in [\![(\_ \rho \_)]\!]^{\mathcal{E}}$
$\rho\ x$	$[\![x]\!]^{\mathcal{E}} \in [\![(\rho \_)]\!]^{\mathcal{E}}$

Note: The infix relation predicate $x \rho y$ is true if the expression x is related to the expression y by the relation $(\_\rho\_)$, i.e. if the tuple (x, y) is a member of the relation $(\_\rho\_)$.

Note: The prefix relation predicate ρx is true if $(\rho\_)$ is true for x, i.e. if x is a member of the set $(\rho\_)$.

Type A predicate $x \in y$ is well-typed if and only if the type of the expression y is the power set type of that of the expression x.

$$[\![x \in y]\!]^{\mathcal{T}} = \mathrm{dom}([\![x]\!]^{\mathcal{T}} ; powerT \cap [\![y]\!]^{\mathcal{T}})$$

Value A predicate $x \in y$ is ZF-true in exactly those environments in which the value of the expression x is a member of the value of the expression y.

$$[\![x \in y]\!]^{\mathcal{V}} = \mathrm{dom}([\![x]\!]^{\mathcal{V}} \cap [\![y]\!]^{\mathcal{V}} ; \ni)$$

8.4 Truth literal

The truth literal **true** represents the predicate that is always true.

Abstract syntax

$$\mathsf{TRUTH} = \mathbf{true}$$

Concrete form

```
TRUE
```

Sample representation and transformation

Representation	Abstract
true	**true**

Type The truth literal **true** is well-typed in all type-environments.

$$[\![\mathit{true}]\!]^{\tau} = \mathit{Tenv}$$

Value The truth literal **true** is ZF-true in all environments.

$$[\![\mathit{true}]\!]^{\nu} = \mathit{Env}$$

8.5 False literal

The false literal **false** represents the predicate that is never true.

Abstract syntax

FALSEHOOD = **false**

Concrete form

FALSE

Sample representation and transformation

Representation	Abstract
false	**false**

Type The false literal **false** is well-typed in all type-environments.

$$\{\!|\, false \,|\!\}^{\tau} = Tenv$$

Value The false literal **false** is not ZF-true in any environment.

$$\{\!|\, false \,|\!\}^{\nu} = \varnothing$$

8.6 Negation

The negation $\neg P$ is true if and only if the predicate P is not.

Abstract syntax A negation is constructed from a predicate.

$$\mathsf{NEGATION} = \neg\, \mathsf{PRED}$$

Concrete form

```
NOT Predicate
```

Sample representation and transformation

Representation	Abstract
$\neg P$	$\neg [\![P]\!]^{\mathcal{P}}$

Type The negation $\neg P$ is well-typed exactly when the predicate P is well-typed.

$$[\![\neg\, P]\!]^{\mathcal{T}} = [\![P]\!]^{\mathcal{T}}$$

Value The negation $\neg P$ is ZF-true in those environments in which the predicate P is not ZF-true.

$$[\![\neg\, P]\!]^{\nu} = Env \setminus [\![P]\!]^{\nu}$$

8.7 Disjunction

The disjunction $P \lor Q$ is true if and only if at least one of the predicates P and Q is true.

Abstract syntax A disjunction is constructed from two predicates.

DISJUNCTION = PRED $\lor$ PRED

Concrete form

`Predicate DISJ Predicate`

Editor's note: This production seems to have been omitted in the Concrete Syntax document. JEN

Sample representation and transformation

Representation	Abstract
$P \lor Q$	$[\![P]\!]^{\mathcal{P}} \lor [\![Q]\!]^{\mathcal{P}}$

Type The disjunction $P \lor Q$ is well-typed in exactly those type-environments in which both predicates P and Q are well-typed.

$$\{\!|P \lor Q|\!\}^{\mathcal{T}} = \{\!|P|\!\}^{\mathcal{T}} \cap \{\!|Q|\!\}^{\mathcal{T}}$$

Value The disjunction $P \lor Q$ is ZF-true in exactly those environments in which one or both of the predicates P , Q are ZF-true.

$$\{\!|P \lor Q|\!\}^{\nu} = \{\!|P|\!\}^{\nu} \cup \{\!|Q|\!\}^{\nu}$$

8.8 Conjunction

The conjunction $P \wedge Q$ is true if both the predicates P and Q are true.

Abstract syntax A conjunction is constructed from two predicates.

$$\mathsf{CONJUNCTION} = \mathsf{PRED} \ \wedge \ \mathsf{PRED}$$

Concrete form

```
Predicate CONJ Predicate
```

Sample representation and transformation There are three ways of constructing a conjunction: by a simple conjunction, by a compound relation, and by separating two or more predicates.

Representation	Abstract
$P \wedge Q$	$[\![P]\!]^{\mathcal{P}} \wedge [\![Q]\!]^{\mathcal{P}}$
$x_1 \ \rho_1 \ x_2 \ \rho_2 \ \ldots \rho_{n-1} \ x_n$	$[\![x_1 \ \rho_1 \ x_2]\!]^{\mathcal{P}} \wedge [\![x_2 \ \rho_2 \ \ldots \rho_{n-1} \ x_n]\!]^{\mathcal{P}}$
$P_1 \mathsf{Sep} \ldots \mathsf{Sep} P_n$	$[\![P_1]\!]^{\mathcal{P}} \wedge \ldots \wedge [\![P_n]\!]^{\mathcal{P}}$

Note: In predicates Sep is a conjunction; such a conjunction has the lowest possible precedence and is equivalent to parenthesising the separate predicates and conjoining them.

Note: Generic emtyset problem.

Editor's note: Review this section!

Type The conjunction $P \wedge Q$ is well-typed in exactly those type-environments in which both the predicates P and Q are well-typed.

$$\{\!|P \wedge Q|\!\}^{\mathcal{T}} = \{\!|P|\!\}^{\mathcal{T}} \cap \{\!|Q|\!\}^{\mathcal{T}}$$

Value The conjunction of two predicates $P \wedge Q$ is ZF-true in exactly those environments in which both the predicates P and Q are ZF-true.

$$\{\!|P \wedge Q|\!\}^{\mathcal{V}} = \{\!|P|\!\}^{\mathcal{V}} \cap \{\!|Q|\!\}^{\mathcal{V}}$$

8.9 Implication

The implication $P \Rightarrow Q$ is true if and only if the predicate P is false or the predicate Q is true.

Abstract syntax An implication is constructed from two predicates.

$$\mathsf{IMPLICATION} = \mathsf{PRED} \Rightarrow \mathsf{PRED}$$

Concrete form

`Predicate IMPLIES Predicate`

Sample representation and transformation

Representation	Abstract
$P \Rightarrow Q$	$[\![P]\!]^{\mathcal{P}} \Rightarrow [\![Q]\!]^{\mathcal{P}}$

Type The implication $P \Rightarrow Q$ is well-typed in exactly those type-environments in which both the predicates P and Q are well-typed.

$$\{\!|P \Rightarrow Q|\!\}^{\tau} = \{\!|P|\!\}^{\tau} \cap \{\!|Q|\!\}^{\tau}$$

Value The implication $P \Rightarrow Q$ is true in exactly those environments in which the predicate P is not ZF-true or the predicate Q is ZF-true.

$$\{\!|P \Rightarrow Q|\!\}^{\nu} = (Env \setminus \{\!|P|\!\}^{\nu}) \cup \{\!|Q|\!\}^{\nu}$$

8.10 Equivalence

An equivalence $P \Leftrightarrow Q$ is true if and only if both predicates P and Q are true or neither is true.

Abstract syntax An equivalence is constructed from two predicates.

$$\mathsf{EQUIVALENCE} = \mathsf{PRED} \Leftrightarrow \mathsf{PRED}$$

Concrete form

```
Predicate IFF Predicate
```

Sample representation and transformation

Representation	Abstract
$P \Leftrightarrow Q$	$[\![P]\!]^{\mathcal{P}} \Leftrightarrow [\![Q]\!]^{\mathcal{P}}$

Type The equivalence $P \Leftrightarrow Q$ is well-typed in exactly those type-environments in which both the predicates P and Q are well-typed.

$$\{\!|P \Leftrightarrow Q|\!\}^{\mathcal{T}} = \{\!|P|\!\}^{\mathcal{T}} \cap \{\!|Q|\!\}^{\mathcal{T}}$$

Value The equivalence $P \Leftrightarrow Q$ is ZF-true in exactly those environments in which the predicates P and Q are both ZF-true or neither are ZF-true.

$$\{\!|P \Leftrightarrow Q|\!\}^{\mathcal{V}} = \{\!|P|\!\}^{\mathcal{V}} \otimes \{\!|Q|\!\}^{\mathcal{V}}$$

8.11 Universal quantification

The universally quantified predicate $\forall \boldsymbol{St} \bullet \boldsymbol{P}$ is true if the predicate $\boldsymbol{P}$ is true for all possible combinations of values of the components of the schema text $\boldsymbol{St}$.

Abstract syntax A universal quantification is constructed from a schema text and a predicate.

$$\mathsf{UNIVERSALQUANT} = \forall \mathsf{SCHEMA} \bullet \mathsf{PRED}$$

Concrete form

```
FORALL TextOrExpression DOT Predicate
```

Sample representation and transformation

Representation	Abstract
$\forall St \bullet P$	$\forall [\![St]\!]^{\mathcal{ST}} \bullet [\![P]\!]^{\mathcal{P}}$

Type A universal quantification $\forall \boldsymbol{St} \bullet \boldsymbol{P}$ is well-typed in a type-environment if and only if the predicate $\boldsymbol{P}$ is well-typed in that type-environment enriched by the schema text $\boldsymbol{St}$.

$$[\![\forall \boldsymbol{St} \bullet \boldsymbol{P}]\!]^{\mathcal{T}} = \mathrm{dom}([\![\boldsymbol{St}]\!]^{\mathcal{T}} \rhd [\![\boldsymbol{P}]\!]^{\mathcal{T}})$$

Meaning A universal quantification $\forall \boldsymbol{St} \bullet \boldsymbol{P}$ is ZF-true in an environment if and only if the predicate $\boldsymbol{P}$ is ZF-true in all enrichments of that environment by the schema text $\boldsymbol{St}$.

$$[\![\forall \boldsymbol{St} \bullet \boldsymbol{P}]\!]^{\nu} = {}^{\forall}([\![\boldsymbol{St}]\!]^{\mathcal{M}})[\![\boldsymbol{P}]\!]^{\nu}$$

Note: This semantic definition rests on the properties of de Morgan's Laws.

8.12 Existential quantification

The existentially quantified predicate $\exists\, \boldsymbol{St} \bullet \boldsymbol{P}$ is true if the predicate $\boldsymbol{P}$ is true for at least one possible combination of values of the components of the schema text $\boldsymbol{St}$.

Abstract syntax An existential quantification is constructed from a schema text and a predicate.

$$\mathsf{EXISTSQUANT} \;=\; \exists\, \mathsf{SCHEMA} \bullet \mathsf{PRED}$$

Concrete form

```
EXISTS TextOrExpression DOT Predicate
```

Sample representation and transformation

Representation	Abstract
$\exists\, St \bullet P$	$\exists [\![St]\!]^{\mathcal{ST}} \bullet [\![P]\!]^{\mathcal{P}}$

Type An existential quantification $\exists\, \boldsymbol{St} \bullet \boldsymbol{P}$ is well-typed in a type-environment if and only if the predicate $\boldsymbol{P}$ is well-typed in that type-environment enriched by the schema text $\boldsymbol{St}$.

$$\{\!|\exists\, \boldsymbol{St} \bullet \boldsymbol{P}\,|\!\}^{\mathcal{T}} \;=\; \mathrm{dom}(\{\!|\boldsymbol{St}\,|\!\}^{\mathcal{T}} \rhd \{\!|\boldsymbol{P}\,|\!\}^{\mathcal{T}})$$

Value An existential quantification $\exists\, \boldsymbol{St} \bullet \boldsymbol{P}$ is ZF-true in an environment if and only if the predicate $\boldsymbol{P}$ is ZF-true in at least one enrichment of that environment by the schema text $\boldsymbol{St}$.

$$\{\!|\exists\, \boldsymbol{St} \bullet \boldsymbol{P}\,|\!\}^{\mathcal{V}} \;=\; \mathrm{dom}(\{\!|\boldsymbol{St}\,|\!\}^{\mathcal{M}} \rhd \{\!|\boldsymbol{P}\,|\!\}^{\mathcal{V}})$$

8.13 Unique existential quantification

The unique existentially quantified predicate $\exists_1 \, \boldsymbol{St} \bullet \boldsymbol{P}$ is true if the predicate $\boldsymbol{P}$ is true for exactly one possible combination of values of the components of the schema text $\boldsymbol{St}$.

Abstract syntax A unique existential quantification is constructed from a schema text and a predicate.

$$\mathsf{UNIQUEQUANT} \;=\; \exists_1 \, \mathsf{SCHEMA} \bullet \mathsf{PRED}$$

Concrete form

```
EXISTS1 TextOrExpression DOT Predicate
```

Sample representation and transformation

Representation	Abstract
$\exists_1 \, St \bullet P$	$\exists_1 [\![St]\!]^{\mathcal{ST}} \bullet [\![P]\!]^{\mathcal{P}}$

Type A unique existential quantification $\exists_1 \, \boldsymbol{St} \bullet \boldsymbol{P}$ is well-typed in a type-environment if and only if the predicate $\boldsymbol{P}$ is well-typed in that type-environment enriched by the schema text $\boldsymbol{St}$.

$$[\![\exists_1 \, \boldsymbol{St} \bullet \boldsymbol{P}]\!]^{\mathcal{T}} \;=\; \mathrm{dom}([\![\boldsymbol{St}]\!]^{\mathcal{T}} \rhd [\![\boldsymbol{P}]\!]^{\mathcal{T}})$$

Value A unique existential quantification $\exists_1 \, \boldsymbol{St} \bullet \boldsymbol{P}$ is ZF-true in an environment if and only if the predicate $\boldsymbol{P}$ is ZF-true in exactly one enrichment of that environment by the schema text $\boldsymbol{St}$.

$$[\![\exists_1 \, \boldsymbol{St} \bullet \boldsymbol{P}]\!]^{\nu} \;=\; \mathrm{dom}(^{\wedge}([\![\boldsymbol{St}]\!]^{\mathcal{M}} \rhd [\![\boldsymbol{P}]\!]^{\nu}) \fatsemi \{-\}^{-1})$$

8.14 Schema predicate

A schema predicate S is true if and only if the values of the components of the schema S are contained in the environment and their values satisfy the property of the schema.

Abstract syntax A schema predicate is constructed from a schema.

SPRED = SCHEMA

Concrete form

```
??  -- to be defined
```

Sample representation and transformation

Representation	Abstract
S	$[\![S]\!]^{\mathcal{S}}$

Type A schema predicate S is well-typed in a type-environment if and only if the schema S is well-typed and the signature of S is contained in the environment.

$$\{\!|S|\!\}^{\mathcal{T}} = \mathrm{dom}((\!| S |\!)^{\mathcal{T}} \cap \supseteq)$$

Value A schema predicate S is ZF-true in an environment if and only if the environment contains a situation of the schema S.

$$\{\!|S|\!\}^{\mathcal{V}} = \mathrm{dom}((\!| S |\!)^{\mathcal{M}} \cap \supseteq)$$

8.15 Substitution

The predicate $b \odot P$ is true if and only if the predicate P is true in the environment enriched by the binding b.

Abstract syntax A substitution instance is constructed from an expression and a predicate.

PREDSUBSTITUTION = EXP⊙PRED

Concrete form

```
Expression SUBST Predicate
```

Sample representation and transformation

Representation	Abstract
$b \odot P$	$[\![b]\!]^{\mathcal{E}} \odot [\![P]\!]^{\mathcal{P}}$

Type A predicate $b \odot P$ is well-typed in an type-environment if and only if b is well-typed with a schema type and the predicate P is well typed in the environment enriched by the binding.

$$\{\!|b \odot P|\!\}^{\mathcal{T}} = \mathrm{dom}(\langle 1, [\![b]\!]^{\mathcal{T}} ; schemaT^{-1} \rangle ; (\oplus) \rhd \{\!| P |\!\}^{\mathcal{T}})$$

Value The predicate $b \odot P$ is ZF-true in exactly those environments in which the binding b is value-defined and, when enriched by b make the predicate P ZF-true.

$$\{\!|b \odot P|\!\}^{\mathcal{V}} = \mathrm{dom}(\langle 1, [\![b]\!]^{\mathcal{M}} ; \diamond \rangle ; (\oplus) \rhd \{\!| P |\!\}^{\mathcal{V}})$$

Editor's note: Revised versions of the following subsections have been included in Annex F, *The logical theory of Z*.

Free variables

Substitution

□

9 Schema

Notes on this section of the Z Standard

Section title: Schema
Source file: sch.tex
Section editor: this version edited by John Nicholls (pro tem)
Original text by: Stephen Brien
Contributions by: Stephen Brien, ... *(to be added)*
Most recent update: 29th June 1995
Formatted: 3rd July 1995

9.1 Introduction

Editor's note: The following note is taken from the proposal by Rob Arthan (dated 27th June 1992) to allow schemas to be regarded as expressions.

A schema is an expression whose value is a set of bindings.

A schema can be used in the following ways:

as a declaration

as a predicate

as an operand of certain operators which construct schemas from other schemas.

Abstract syntax

$$
\begin{aligned}
\mathsf{SCHEMA} \;=\; & \mathsf{SDECL} \\
| \; & \mathsf{SCONSTRUCTION} \\
| \; & \mathsf{SNEGATION} \\
| \; & \mathsf{SDISJUNCTION} \\
| \; & \mathsf{SCONJUNCTION} \\
| \; & \mathsf{SIMPLICATION} \\
| \; & \mathsf{SEQUIVALENCE} \\
| \; & \mathsf{SPROJECTION} \\
| \; & \mathsf{SHIDING} \\
| \; & \mathsf{SUNIVQUANT} \\
| \; & \mathsf{SEXISTSQUANT} \\
| \; & \mathsf{SUNIQUEQUANT} \\
| \; & \mathsf{SRENAMING} \\
| \; & \mathsf{SCOMPOSITION} \\
| \; & \mathsf{SDECORATION} \\
| \; & \mathsf{SSUBSTITUTION} \\
| \; & \mathsf{EXPSCHEMA}
\end{aligned}
$$

Strategy for definition

When making schemas, the problem is not so much whether it is well defined (although a schema may fail to be defined). The problem is more to record the possible meanings of the declared names. The definition is built up in two stages. The *type* function defines the *signature* of a schema. The *meaning* relation relates the environment to those possible *situations* defined by the schema.

A schema can be also used to introduce new variables to the environment, A type and meaning enrichment function is given for this purpose.

9.1.1 Type function

For any schema S its *type* function is a recursively defined partial function from type-environments to signatures which record the types of the elements denoted by the variables introduced:

$$(\!| S |\!)^{\tau} \;:\; Tenv \pfun Signature$$

The schema S is *well-typed* in exactly those type-environments contained in $\mathrm{dom}\,(\!| S |\!)^{\tau}$. The signature of a schema in a type-environment is the result of applying its type function to that type-environment.

For any schema S its *type enrichment function* is a partial function from a type-environment to a new one in which the names of the constituent schema are known:

$$\{S\}^{\tau} \;:\; Tenv \pfun Tenv$$

$$\{S\}^{\tau} \;=\; \langle 1, (\!| S |\!)^{\tau} \rangle \fatsemi \oplus$$

9.1.2 Meaning relation

A schema introduces names to the environment which can assume certain values. These values are not fixed. We can consider the meaning of a schema as a set of situations, each one recording one set of values for the new names. However, it is more convenient to consider the meaning of a schema as a relation between environments and situations. For any schema S, its *meaning relation* is a relation from environments to situations:

$$(\!|\ S\ |\!)^{\mathcal{M}} : Env \leftrightarrow Situation$$

The meaning of a schema in an environment is any one of the situations related to that environment by the meaning function. The meaning of a schema is partial because some schemas may fail – for example $n : s$ where s is undefined, or if s is an empty set. A schema S is *well-defined* in exactly those environments contained in $\mathrm{dom}\,(\!|\ S\ |\!)^{\mathcal{M}}$.

The meaning enrichment is represented as a relation between environments, for the same reason as the meaning of a schema as represented by a relation.

$$(\!|S\ |\!)^{\mathcal{M}} : Env \leftrightarrow Env$$

$$(\!|S\ |\!)^{\mathcal{M}} = \langle 1, (\!|\ S\ |\!)^{\mathcal{M}} \rangle \fatsemi \oplus$$

9.2 Schema designator

A schema designator is a schema name used to refer to schema. It may also contain a list of generic paramaters which instantiate a generically defined schema.

> **Note:** Since schema names have global scope there cannot be any overlap between the base names of variables and schema names in a specification.

Abstract syntax A schema designator is constructed from a schema name.

SDES = WORD

Concrete form

`Nofix`

Sample representation and transformation

Representation	Abstract
S	$\boldsymbol{S}$

Type The signature of a schema reference is the signature of the type of the reference in the type-environment.

$$(\!| S |\!)^{\mathcal{T}} = (1 \bullet S^{\circ}) \fatsemi power\mathcal{T}^{-1} \fatsemi schema\mathcal{T}^{-1}$$

> **Note:** A schema reference is well-typed only if it is in the domain of the type-environment.

Meaning The meaning of a schema reference is the relation constructed from the meaning of the reference in the environment.

$$(\!| S |\!)^{\mathcal{M}} = (1 \bullet S^{\circ}) \fatsemi \ni$$

> **Note:** A schema reference is well-defined only if it is in the domain of the environment.

9.3 Generic schema designator

A generic schema designator $S[x_1, \ldots, x_n]$ is reference to a generically defined schema S instantiated by the set paramaters $[x_1, \ldots, x_n]$.

Abstract syntax A generic schema designator is constructed from a schema name and a list of expressions.

$$\mathsf{SGENDES} = \mathsf{WORD}\ [\mathsf{EXP}, \ldots, \mathsf{EXP}]$$

Concrete form

```
NAME SQBRA ExpressionList1 SQKET
```

Sample representation and transformation

Representation	Abstract
$S_{[x_1,\ldots,x_n]}$	$S[(\!\|x_1\|\!)^{\mathcal{E}}, \ldots, (\!\|x_n\|\!)^{\mathcal{E}}]$

Type

$$(\!|\ S[x_1, \ldots, x_n]\ |\!)^{\mathcal{T}} = ((1 \bullet S^{\circ}) \bullet \langle x_1, \ldots, x_n \rangle) \fatsemi power T^{-1} \fatsemi schema T^{-1}$$

Meaning

$$(\!|\ S[x_1, \ldots, x_n]\ |\!)^{\mathcal{M}} = ((1 \bullet S^{\circ}) \bullet \langle x_1, \ldots, x_n \rangle) \fatsemi \ni$$

Note: Generically defined schemas must be instantiated.

9.4 Simple schema

A simple schema $n_1, \ldots n_m : s$ introduces variables named $n_1, \ldots n_m$ whose values are drawn from the set s.

Abstract syntax A simple schema is constructed from a list of names and an expression.

SDECL = NAME, NAME, ..., NAME : EXP

Concrete form

```
NameList1 COLON Expression
```

Sample representation and transformation

Representation	Abstract
$n_1, n_2, \ldots, n_m : s$	$n_1, n_2, \ldots, n_k : [\![s]\!]^{\mathcal{E}}$

Type The type of the simple schema $n_1, \ldots n_m : s$ is the signature constructed from the names $n_1, \ldots n_m$ and the underlying type of the set expression s.

$$(\!| n_1, \ldots, n_m : s |\!)^{\mathcal{T}} = [\![s]\!]^{\mathcal{T}} ; \langle \langle n_1{}^{\circ}, powerT^{-1} \rangle, \ldots, \langle n_m^{\circ}, powerT^{-1} \rangle \rangle ; \{\ldots\}$$

Note: The simple schema $n_1, \ldots n_m : s$ is well-typed if and only if the expression s has power set type.

Meaning The meaning of the simple schema $n_1, \ldots n_m : s$ is a relation from the environment to those situations which associate each of the names $n_1, \ldots n_m$ with one of the elements of the set expression s:

$$(\!| n_1, \ldots n_m : s |\!)^{\mathcal{M}} = [\![s]\!]^{\mathcal{M}} ; \langle \langle n_1{}^{\circ}, \exists \rangle, \ldots, \langle n_m{}^{\circ}, \exists \rangle \rangle ; \{\ldots\}$$

Note: The simple schema $n_1, \ldots n_m : s$ is well-defined if and only if the expression s is a non-empty set.

Note: Suppose G is defined to be a given set. The type system defines the type of G to be $powerT(givenT\,\mathbb{N})$. In this way a schema such as $x : G$ defines the type of x to be $givenT(G)$, as required.

9.5 Schema construction

A schema construction $S \mid P$ is a schema whose signature is that of the schema S and whose components satisfy the constraint of the schema S and the predicate P.

Abstract syntax A schema construction is composed from a schema and a predicate.

SCONSTRUCTION = SCHEMA | PRED

Concrete form

```
DeclPart [VBAR Predicate]
SQBRA Text SQKET
```

Sample representation and transformation

Representation	Abstract
$D \mid P$	$[\![D]\!]^{\mathcal{D}} \mid [\![P]\!]^{\mathcal{P}}$
$[D \mid P]$	$\langle [\![D]\!]^{\mathcal{D}} \mid [\![P]\!]^{\mathcal{P}} \rangle$
$[D]$	$\langle [\![D]\!]^{\mathcal{D}} \mid \mathbf{true} \rangle$

Type The signature of $\langle D \mid P \rangle$ is the same as that of the schema D.

$$(\!| S \mid P \rangle |\!)^{\mathcal{T}} = (\!| S |\!)^{\mathcal{T}} \cap ((\!| D \mid P |\!)^{\mathcal{T}} ; \supseteq)$$

Meaning The value of the schema expression constructed from $\langle D \mid P \rangle$ is a set of bindings. The bindings are constructed in all enrichments of the environment by D which satisfy P:

$$(\!| \langle D \mid P \rangle |\!)^{\mathcal{M}} = (\!| D |\!)^{\mathcal{M}} \cap ((\!| D \mid P |\!)^{\mathcal{M}} ; \supseteq)$$

This is defined only in those environments in which the schema D is defined and when enriched by it result in the predicate P being well-typed.

9.6 Schema negation

A schema negation $\neg S$ is a schema which contains all the bindings of the same signature as those of the schema S but which are not contained in S.

Abstract syntax A schema negation is composed of a schema

$$\mathsf{SNEGATION} = \neg\, \mathsf{SCHEMA}$$

Concrete form

`NOT Expression`

Sample representation and transformation

Representation	Abstract
$\neg S$	$\neg (\!\| S \|\!)^{\mathcal{S}}$

Type The signature of a negated schema $\neg S$ is the same signature as that of the schema S:

$$(\!| \neg S |\!)^{\mathcal{T}} = (\!| S |\!)^{\mathcal{T}}$$

Meaning The bindings of a negated schema $\neg S$ are those bindings which have the same signature as S but are not bindings of S:

$$(\!| \neg S |\!)^{\mathcal{M}} = (\!| S |\!)^{\mathcal{MT}} \setminus (\!| S |\!)^{\mathcal{M}}$$

Note: This is simpler than in (Spivey, 1988), where this complement had to be combined with the global part of the environment. This was necessary in the original semantics, because the meaning of a schema involved not only the components of the schema, but also the global variables to which the schema might refer.

9.7 Schema disjunction

The schema disjunction $S_1 \lor S_2$ is a schema whose signature is the join of the signatures of the two schemas S_1 and S_2 and whose property is the disjunction of the two schemas' properties.

Abstract syntax A schema disjunction is composed of two schemas.

SDISJUNCTION = SCHEMA $\lor$ SCHEMA

Concrete form

`Expression OR Expression`

Sample representation and transformation

Representation	Abstract
$S_1 \lor S_2$	$[\![S_1]\!]^{\mathcal{S}} \lor [\![S_2]\!]^{\mathcal{S}}$

Type The signature of a schema disjuinction $S_1 \lor S_2$ is the join of the two schemas S_1 and S_2 :

$$(\!| S_1 \lor S_2 |\!)^{\mathcal{T}} = \langle (\!| S_1 |\!)^{\mathcal{T}}, (\!| S_2 |\!)^{\mathcal{T}} \rangle \, ; \sqcup$$

Note: The schema disjunction $S_1 \lor S_2$ is well-typed only if the signature of the two schemas S_1 and S_2 are type compatible.

Meaning The bindings of a disjoined schema are all those with its signature which are extensions of bindings in one or other of the operand schemas:

$$(\!| S_1 \lor S_2 |\!)^{\mathcal{M}} = (\langle (\!| S_1 |\!)^{\mathcal{MT}}, (\!| S_2 |\!)^{\mathcal{M}} \rangle \cup \langle (\!| S_1 |\!)^{\mathcal{M}}, (\!| S_2 |\!)^{\mathcal{MT}} \rangle) \, ; \sqcup$$

9.8 Schema conjunction

Abstract syntax A schema conjunction is composed of two schemas

SCONJUNCTION = SCHEMA $\wedge$ SCHEMA

Concrete form

`DeclElem SEMICOLON DeclElem {SEMICOLON DeclElem}`

`Expression AND Expression`

Sample representation and transformation

Representation	Abstract
$D_1; D_2; \ldots; D_n$	$[\![D_1]\!]^{\mathcal{D}}; [\![D_2]\!]^{\mathcal{D}}; \ldots; [\![D_n]\!]^{\mathcal{D}}$
$S_1 \wedge S_2$	$[\![S_1]\!]^{\mathcal{S}} \wedge [\![S_2]\!]^{\mathcal{S}}$

Variables may be introduced in local schemas more than once, provided that they have the same type. Repeated schemas do not add anything to the signature; however the constraint of the repeated schema is conjoined with the constraints of all the other schemas.

Type The signature of a schema conjunction $S_1 \wedge S_2$ is the join of the two schemas S_1 and S_2:

$$(\!|\, S_1 \wedge S_2 \,|\!)^{\mathcal{T}} = \langle (\!|\, S_1 \,|\!)^{\mathcal{T}}, (\!|\, S_2 \,|\!)^{\mathcal{T}} \rangle \,;\, \sqcup$$

Note: The schema conjunction $S_1 \wedge S_2$ is well-typed only if the two schemas S_1 and S_2 are well-typed and their signatures are type compatible.

Meaning The bindings of a conjoined schema are all those with its signature which are extensions of bindings in both of the operand schemas:

$$(\!|\, S_1 \wedge S_2 \,|\!)^{\mathcal{M}} = \langle (\!|\, S_1 \,|\!)^{\mathcal{M}}, (\!|\, S_2 \,|\!)^{\mathcal{M}} \rangle \,;\, \sqcup$$

Note: Spivey (1988) has already remarked on the similarity with the semantics of the parallel composition operator in the traces model of CSP.

Note: Duplicated schemas are significant in the evaluation of the characteristic tuple. The representative term can be a list of terms which form part of the top level tuple.

9.9 Schema implication

Abstract syntax A schema implication is composed of two schemas.

SIMPLICATION = SCHEMA ⇒ SCHEMA

Concrete form

`Expression IMPLIES Expression`

Representation	Abstract
$S_1 \Rightarrow S_2$	$[\![S_1]\!]^{\mathcal{S}} \Rightarrow [\![S_2]\!]^{\mathcal{S}}$

Type The signature of a schema implication $\boldsymbol{S}_1 \Rightarrow \boldsymbol{S}_2$ is the join of the two schemas $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$:

$$(\!|\, \boldsymbol{S}_1 \Rightarrow \boldsymbol{S}_2 \,|\!)^{\mathcal{T}} = \langle (\!|\, \boldsymbol{S}_1 \,|\!)^{\mathcal{T}}, (\!|\, \boldsymbol{S}_2 \,|\!)^{\mathcal{T}} \rangle \, ; \sqcup$$

Note: The schema implication $\boldsymbol{S}_1 \Rightarrow \boldsymbol{S}_2$ is well-typed only if the two schemas $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$ are well-typed and their signatures are type compatible.

Meaning The meaning of the schema implication $\boldsymbol{S}_1 \Rightarrow \boldsymbol{S}_2$ is the same as the meaning of the schema disjunction $\neg \boldsymbol{S}_1 \lor \boldsymbol{S}_2$:

$$(\!|\, \boldsymbol{S}_1 \Rightarrow \boldsymbol{S}_2 \,|\!)^{\mathcal{M}} = (\!|\, \neg \boldsymbol{S}_1 \lor \boldsymbol{S}_2 \,|\!)^{\mathcal{M}}$$

9.10 Schema equivalence

Abstract syntax A schema equivalence is composed of two schemas.

$$\mathsf{SEQUIVALENCE} = \mathsf{SCHEMA} \Leftrightarrow \mathsf{SCHEMA}$$

Concrete form

```
Expression IFF Expression
```

Sample representation and transformation

Representation	Abstract
$S_1 \Leftrightarrow S_2$	$[\![S_1]\!]^S \Leftrightarrow [\![S_2]\!]^S$

Type The signature of a schema equivalence $S_1 \Leftrightarrow S_2$ is the join of the two schemas S_1 and S_2 :

$$(\!| S_1 \Leftrightarrow S_2 |\!)^\mathcal{T} = \langle (\!| S_1 |\!)^\mathcal{T}, (\!| S_2 |\!)^\mathcal{T} \rangle \fatsemi \sqcup$$

Note: The schema equivalence $S_1 \Leftrightarrow S_2$ is well-typed only if the two schemas S_1 and S_2 are well-typed and their signatures are type compatible.

Meaning The bindings are all those with this signature which are extensions of bindings in neither or both of the operand schema expressions:

$$(\!| S_1 \Leftrightarrow S_2 |\!)^\mathcal{M} = (\!| S_1 \Rightarrow S_2 \land S_2 \Rightarrow S_1 |\!)^\mathcal{M}$$

9.11 Schema projection

The schema projection operator ($\upharpoonright$) hides all the components of its first argument except those which are also components of its second argument.

Abstract syntax A schema projection is composed of two schemas.

SPROJECTION = SCHEMA **Proj** SCHEMA

Concrete form

`Expression PROJECTION Expression`

Sample representation and transformation

Representation	Abstract
$S \upharpoonright T$	$[\![S]\!]^{\mathcal{S}} \upharpoonright [\![T]\!]^{\mathcal{S}}$

Type The signature of a projection $S_1 \upharpoonright S_2$ includes those names in both the domains of the signatures of S_1 and S_2. The type given to each such name is taken from S_1. Note that if names are given types by both S_1 and S_2 those types must be the same (that is, the signatures must be consistent):

$$(\!|\, S_1 \upharpoonright S_2 \,|\!)^{\mathcal{T}} = \langle (\!|\, S_1 \,|\!)^{\mathcal{T}}, (\!|\, S_2 \,|\!)^{\mathcal{T}} \rangle \,;\, \sqcap$$

Meaning The value of the projection $S_1 \upharpoonright S_2$ is the set of bindings which satisfy S_1, restricted to the alphabet of S_2:

$$(\!|\, S_1 \upharpoonright S_2 \,|\!)^{\mathcal{M}} = \langle (\!|\, S_1 \,|\!)^{\mathcal{M}}, (\!|\, S_2 \,|\!)^{\mathcal{MT}} \rangle \,;\, \sqcap$$

Note: Spivey (1988) gives two forms of projection operator used in a schema expression such as $S_1 \upharpoonright S_2$. The weak operator hides those components of S_1 which are not in the signature of S_2. The strong form requires the components to satisfy the axioms of S_2 as well. We give the semantics for the weak operator.

9.12 Schema hiding

The hiding operator (\) takes a schema expression as its first operand and an identifier list as its second operand. The result is a schema expression whose components are those of the operand schema excluding those named in the list.

Abstract syntax A hidden schema is composed of a schema and a list of names.

SHIDING = SCHEMA \ [NAME,...,NAME]

Concrete form

`Expression HIDING BRA NameList1 KET`

Sample representation and transformation

Representation	Abstract
$S \setminus (n_1, n_2, \ldots, n_m)$	$[\![S]\!]^{\mathcal{S}} \setminus < n_1, n_2, \ldots, n_m >$

Type The signature of a schema hiding expression is the signature of S with the names from $(n_1, \ldots, n_m)$ removed. Note that $(n_1, \ldots, n_m)$ may contain names not in the signature of *se*:

$$(\!| S \setminus (n_1, \ldots, n_m) |\!)^{\mathcal{T}} = (\!| S |\!)^{\mathcal{T}} ; (\{n_1, \ldots, n_m\} \lhd)$$

Meaning The value of the schema S in which the components $(n_1, \ldots, n_m)$ have been hidden is the set of bindings which satisfy S, with those components removed:

$$(\!| S \setminus (n_1, \ldots n_m) |\!)^{\mathcal{M}} = (\!| S |\!)^{\mathcal{M}} ; (\{n_1, \ldots n_m\} \lhd)$$

Note: If all the variables are hidden the result is a schema with an empty signature.

9.13 Schema universal quantification

Abstract syntax A schema quantification is constructed from a schema text and a schema.

$$\mathsf{SUNIVQUANT} = \forall \mathsf{SCHEMA} \bullet \mathsf{SCHEMA}$$

Concrete form

```
EXISTS TextOrExpression DOT Expression
```

Sample representation and transformation

Representation	Abstract
$\forall St \bullet S$	$\forall [\![St]\!]^{\mathcal{ST}} \bullet [\![S]\!]^{\mathcal{S}}$

Type The signature of a universally quantified schema expression $\forall \boldsymbol{St} \bullet \boldsymbol{S}$ is the signature of $\boldsymbol{S}$ with the names from the signature of $\boldsymbol{St}$ removed:

$$(\!|\, \forall \boldsymbol{St} \bullet \boldsymbol{S} \,|\!)^{\mathcal{T}} = \langle (\!|\, \boldsymbol{S} \,|\!)^{\mathcal{T}}, (\!|\, \langle \boldsymbol{St} \rangle \,|\!)^{\mathcal{T}} \rangle \; ;\!\leftarrow$$

Note: The signature is well-typed only when $\boldsymbol{St}$ and $\boldsymbol{S}$ is are well-typed and their signatures are compatible.

Meaning The value of a universally quantified schema expression $\forall \boldsymbol{St} \bullet \boldsymbol{S}$ is the set of bindings with the defined signature such that, for all bindings of $\boldsymbol{St}$, the union of the two bindings is an extension of $\boldsymbol{S}$:

$$(\!|\, \forall \boldsymbol{St} \bullet \boldsymbol{S} \,|\!)^{\mathcal{M}} = (\!|\, \neg \exists \boldsymbol{St} \bullet \neg \boldsymbol{S} \,|\!)^{\mathcal{M}}$$

Note: Note that this definition takes advantage of de Morgan's Law.

9.14 Schema existential quantification

Abstract syntax A schema quantification is composed of a schema text and a schema.

$$\mathsf{SEXISTSQUANT} = \exists\, \mathsf{SCHEMA} \bullet \mathsf{SCHEMA}$$

Concrete form

```
EXISTS TextOrExpression DOT Expression
```

Sample representation and transformation

Representation	Abstract
$\exists\, St \bullet S$	$\exists [\![St]\!]^{\mathcal{ST}} \bullet [\![S]\!]^{\mathcal{S}}$

Type The signature of an existentially quantified schema expression $\exists\, \boldsymbol{St} \bullet \boldsymbol{S}$ is the signature of S with the names from the signature of St removed:

$$(\!|\, \exists\, \boldsymbol{St} \bullet \boldsymbol{S} \,|\!)^{\mathcal{T}} = \langle (\!|\, \boldsymbol{S} \,|\!)^{\mathcal{T}}, (\!|\, \langle \boldsymbol{St} \rangle \,|\!)^{\mathcal{T}} \rangle \,;\!\!\leftarrow$$

Note: The signature is well-typed only when $\boldsymbol{St}$ and $\boldsymbol{S}$ is are well-typed and their signatures are compatible.

Meaning The value of an existentially quantified schema expression $\exists\, \boldsymbol{St} \bullet \boldsymbol{S}$ is the set of bindings with signature of $\boldsymbol{S}$ less $\boldsymbol{St}$, such that there is a binding of $\boldsymbol{St}$ so that the union of the two bindings is an extension of $\boldsymbol{S}$:

$$(\!|\, \exists\, \boldsymbol{St} \bullet \boldsymbol{S} \,|\!)^{\mathcal{M}} = \langle (\!|\, \boldsymbol{S} \,|\!)^{\mathcal{M}}, (\!|\, \langle \boldsymbol{St} \rangle \,|\!)^{\mathcal{M}} \rangle \,;\!\!\leftarrow$$

Note: This definition should be contrasted with the analogous expression for predicates $(\exists\, St \bullet p)$ where the well-typing of the predicate is decided in the modified environment.

9.15 Schema unique existential quantification

Abstract syntax A schema quantification is composed of a schema text and a schema.

$$\mathsf{SUNIQUEQUANT} = \exists_1 \mathsf{SCHEMA} \bullet \mathsf{SCHEMA}$$

Concrete form

```
EXISTS1 TextOrExpression DOT Expression
```

Sample representation and transformation

Representation	Abstract
$\exists_1 St \bullet S$	$\exists_1 [\![St]\!]^{\mathcal{ST}} \bullet [\![S]\!]^{\mathcal{S}}$

Type

$$(\!| \exists_1 St \bullet S |\!)^{\mathcal{T}} = \langle (\!| S |\!)^{\mathcal{T}}, (\!| \langle St \rangle |\!)^{\mathcal{T}} \rangle \; ;\!\leftarrow$$

Note: The signature is well-typed only when St and S is are well-typed and their signatures are compatible.

Meaning The value of an existentially quantified schema expression $\exists_1 St \bullet S$ is the set of bindings with signature of S less St, such that there exists a unique binding of St so that the union of the two bindings is an extension of S:

$$(\!| \exists_1 St \bullet S |\!)^{\mathcal{M}} = \text{To be defined}$$

9.16 Schema renaming

The renaming operation $S[new/old]$ substitutes the new variable name for the old in the schema.

Abstract syntax A schema renaming consists of a schema and a renaming list.

SRENAMING = SCHEMA [NAME/NAME, ..., NAME/NAME]

Concrete form

`Expression SQBRA RenameList SQKET`

Sample representation and transformation

Representation	Abstract
$S\,[x_1/y_1, x_2/y_2, \ldots x_n/y_n]$	$[\![S]\!]^{\mathcal{S}} < x_1/y_1, x_2/y_2, \ldots x_n/y_n >$

Type Schema renaming changes the names of the elements in the bindings, and hence the signature.

$$(\!|\, S[Nl] \,|\!)^{\mathcal{T}} = (\!|\, S \,|\!)^{\mathcal{T}} \fatsemi \exists((\!|\, Nl \,|\!)^{\mathcal{N}} \otimes 1)$$

Meaning

$$(\!|\, S[Nl] \,|\!)^{\mathcal{M}} = (\!|\, S \,|\!)^{\mathcal{M}} \fatsemi \exists((\!|\, Nl \,|\!)^{\mathcal{N}} \otimes 1)$$

Note: When more than one variable is to be substituted, the substitution is simultaneous. Any substitutions for non-existent names are ignored. Each old name can only be substituted by one new name. Likewise, each new name can be a substitute for only one old name.

9.17 Substituted schema

The meaning of the substituted schema $b \odot S$ is the same as the meaning of the schema S in the environment enriched by the binding b.

Abstract syntax A substituted schema is composed of an expression and a schema.

$$\mathsf{SSUBSTITUTION} = \mathsf{EXP} \odot \mathsf{SCHEMA}$$

Concrete form

Expression , '$\odot$' , Schema

Sample representation and transformation

Production	Representation	Abstract
$b \odot S$	$[\![b]\!]^{\mathcal{E}} \odot [\![S]\!]^{\mathcal{S}}$	
$b \odot St$	$[\![b]\!]^{\mathcal{E}} \odot [\![St]\!]^{\mathcal{ST}}$	
$b \odot D$	$[\![b]\!]^{\mathcal{E}} \odot [\![D]\!]^{\mathcal{D}}$	

Type The signature of the substituted schema $b \odot S$ is the signature of the schema S in the type-environment enriched by the binding b.

$$(\!|\, b \odot S \,|\!)^{\mathcal{T}} = \langle 1, [\![\, b \,]\!]^{\mathcal{T}} \fatsemi schemaT^{-1} \rangle \fatsemi \oplus \fatsemi (\!|\, S \,|\!)^{\mathcal{T}}$$

A substituted schema is well-typed if and only if the binding is well-typed and the schema is well-typed in the enriched environment.

Meaning The situations of the substituted schema $b \odot S$ are the situations of the schema S in the environment enriched by the binding b.

$$(\!|\, b \odot S \,|\!)^{\mathcal{M}} = \langle 1, [\![\, b \,]\!]^{\mathcal{M}} \fatsemi \langle \_, \_ \rangle \rangle \fatsemi \oplus \fatsemi (\!|\, S \,|\!)^{\mathcal{M}}$$

Editor's note: Revised versions of the following subsections have been included in Annex F, *The logical theory of Z.*

Free variables

Substitution

□

10 Paragraph

Notes on this section of the Z Standard

Section title: Paragraph
Section editor: Peter Lupton (this version edited by JEN)
Original text by: Stephen Brien
Contributions by: Stephen Brien, ... *(others to be added)*
Source file: par.tex
Most recent update: 21st June 1995
Formatted: 3rd July 1995

Editor's note: This is a revision of the Paragraph section, incorporating the proposed new Concrete Syntax in a provisional form. The section will be updated and revised by Peter Lupton.

10.1 Introduction

Each paragraph of Z can do two things: Augment the environment by a declaration and strengthen the property by a predicate. Each paragraph is considered as a relation between environments. The domain of this relation contains all the environments in which the paragraph is well-typed and any predicates contained within it are true. These environments are related to those which include the new variables declared in their signature and which satisfy any property denoted by the paragraph.

$$(\!|\, \mathsf{PAR} \,|\!)^{\mathcal{T}} \;:\; \mathit{Tenv} \pfun \mathit{Tenv}$$

$$(\!|\, \mathsf{PAR} \,|\!)^{\mathcal{M}} \;:\; \mathit{Env} \rightarrow \mathit{Env}$$

We can prove the following

$$\vdash (\!|\, \mathit{Par} \,|\!)^{\mathcal{M}} \fatsemi \Upsilon \subseteq \Upsilon \fatsemi (\!|\, \mathit{Par} \,|\!)^{\mathcal{T}}$$

Abstract Syntax

$$\begin{aligned} \mathsf{PAR} = {} & \mathsf{GIVENSETDEF} \\ | \; & \mathsf{GLOBALPRED} \\ | \; & \mathsf{GLOBALSCHEMA} \\ | \; & \mathsf{GENERICSCHEMA} \\ | \; & \mathsf{GLOBALDEF} \\ | \; & \mathsf{GENERICDEF} \\ | \; & \mathsf{CONJECTURE} \end{aligned}$$

10.2 Given sets

The given set definition $[X_1, X_2, \ldots, X_n]$ introduces the sets $X_1, X_2, \ldots, X_n$ without determining their elements.

Note: Distinctly named given sets have distinct types and hence are incomparable.

Abstract syntax

$$\mathsf{GIVENSETDEF} = \textbf{given } [\mathsf{NAME}, \mathsf{NAME}, \ldots, \mathsf{NAME}]$$

Concrete form

```
SQBRA NameList1 SQKET
```

Sample representation and transformation

Representation	Abstract
$[X_1, X_2, \ldots, X_n]$	$\textbf{given } \langle X_1, \ldots, X_n \rangle$

Type The declaration of given sets $\textbf{given}[X_1, \ldots, X_n]$ causes the type environment to be suitably enriched. Each name is given the power set type of the given type of that name. These declarations over-ride the environment.

Note that a given set definition of $\mathbb{N}$ results in $\mathbb{N}$ having the type $powerT\ givenT\ \mathbb{N}$.

$$(\!|\textbf{given}\langle X_1, \ldots, X_n \rangle |\!)^{\mathcal{T}} = \langle 1, (\{X_1, \ldots, X_n\} \lhd givenT \fatsemi powerT)^\circ \rangle \fatsemi \oplus$$

Meaning To enrich the meaning environment, we construct a binding of the given set names (those in $\operatorname{ran} s$) to typed values in the world of sets – for this to be correct, the bindings must be such that the given sets do indeed have power set type. The environment is updated with this binding.

$$(\!|\textbf{given}\langle X_1, \ldots, X_n \rangle |\!)^{\mathcal{M}} = \langle 1, (\{X_1, \ldots, X_n\} \lhd givenT \fatsemi \langle powerT, Carrier \rangle)^\circ \rangle \fatsemi \oplus$$

10.3 Constraint

A constraint is a predicate appearing on its own as a paragraph. It denotes a property of the values of variables declared elsewhere with global scope. This property is conjoined to the global property.

Abstract syntax

GLOBALPRED = **where** PRED

Concrete form

`Predicate`

Sample representation and transformation

Representation	Abstract
P	**where** $(\!\lvert P \rvert\!)^{\mathcal{P}}$

Type A constraint adds nothing to the environment, so it is that subset of the identity relation restricted to the environments in which the predicate is true.

For the type environment:

$$(\!\lvert P \rvert\!)^{\mathcal{T}} = {}^{1}\!\!\lhd (\!\lvert P \rvert\!)^{\mathcal{T}}$$

Meaning For meaning environment:

$$(\!\lvert P \rvert\!)^{\mathcal{M}} = {}^{1}\!\!\lhd (\!\lvert P \rvert\!)^{\mathcal{M}}$$

10.4 Global declaration

An axiomatic definition introduces variables and specifies further properties of the elements denoted by them.

Abstract syntax

GLOBALSCHEMA = **def** SCHEMA

Concrete form

AX [DeclPart | Expression] [ST Predicate] END

'AX' , DeclPart , 'END'

Sample representation and transformation

Representation	Abstract
'AX' D 'ST' P 'END'	**def** $[\![D]\!]^{\mathcal{D}}$ | $[\![P]\!]^{\mathcal{P}}$
'AX' D 'END'	**def** $[\![D]\!]^{\mathcal{D}}$ | **true**

The abstract form of an axiomatic definition is a pair of paragraphs, one containing a declaration and the other a predicate. If the AxiomPart is omitted the the abstract form is one declaration paragraph.

Type When new variables are declared the environment is enriched by a function from their names to one from their empty generic parameter list to their meaning. We give as its value a set of bindings, one for each name declared. In obtaining the binding, we enrich the environment with the declaration in such a way that the constraint is satisfied. The names in the declaration are bound to their values in this enriched environment. Formally:

$$(\!|\mathbf{def} D \mid P |\!)^{\mathcal{T}} = (\!|D \mid P |\!)^{\mathcal{T}}$$

Meaning

$$(\!|\mathbf{def} D \mid P |\!)^{\mathcal{M}} = (\!|D \mid P |\!)^{\mathcal{M}}$$

Note: The sets from which the elements denoted by the variables can be drawn are defined by the conjunction of the constraint of the DeclPart and the property in the AxiomPart.

The signature of the DeclPart is joined to the global signature. The constraint in the DeclPart and the property of the AxiomPart are conjoined to the global property.

10.5 Generic declarations

A generic declaration of variables adds these variables to the dictionary and maps them to a function from all possible instantiations of their generic parameters to the values of the variables with these instantiations.

Abstract syntax

GENERICSCHEMA = **gendef** [NAME, NAME, ..., NAME] **const** SCHEMA

Concrete form

```
GEN [Formals] BAR [DeclPart | Expression] [ST Predicate] END
```

'GEN', GenFormals, 'BAR', DeclPart, 'END'

Sample representation and transformation

Representation	Abstract
'GEN' [$X_1, X_2, \ldots, X_n$] 'BAR' D 'ST' P 'END'	**gendef** $\langle X_1, X_2, \ldots, X_n \rangle$ **const** $(\!\| D \|\!)^{\mathcal{D}}$ **where** $(\!\| P \|\!)^{\mathcal{P}}$
'GEN' [$X_1, X_2, \ldots, X_n$] 'BAR' D 'END'	**gendef** $\langle X_1, X_2, \ldots, X_n \rangle$ **const** $(\!\| D \|\!)^{\mathcal{D}}$ **where true**

Type

Value A generic declaration introduces a family of variables, parametrised by the generic parameters of the list GenFormals.

Note: In a GenericDef, the DeclPart declares the names of the generic variables whose types can be determined upon instantiation of the formal parameters. The predicate in the AxiomPart determines the elements denoted by the variables for each value of the formal parameters.

Recursive generic declarations are not allowed. The generic declaration must not place any restriction on the generic parameters.

A generic variable has global scope, excluding the declaration list in which it is declared and any construct in which its name is re-used for a local variable.

The parameters of a generic declaration are local to the declaration, but they can be instantiated by elements of set type when the generic variable is used.

A generic declaration does not give a single type: rather, a function from the generic parameters to types is defined.

Let X and Y be generic formal parameters and consider a generic declaration which declares $x : X; y : Y$. Then an expression such as $x \in y$ or $x = y$ would impose a mutual constraint on the types that could be used to instantiate X and Y. For $x \in y$, we have the constraint that the types that Y may take are the powerset of the types that X may take; for $x = y$, we have the constraint that the types that Y may take must be the same as the types that X may take.

The definition of generic types as total functions imposes the constraint that generic declarations do not create relationships between the type of their formal parameters. Such relationships can always be eliminated within a specification.

Since all the type constructors are bijections, any relationship between the types of generic parameters is functional. Therefore any dependent parameters are redundant since they can be uniquely determined as functions of the other parameters. For instance, for $x \in y$ the relationship can be eliminated by removing Y as a formal generic parameter and defining $y : \mathbb{P} X$; for $x = y$ we can eliminate Y and define $y : X$.

10.6 Global definitions

Abstract syntax

GLOBALDEF = **abbr** NAME := EXP

Concrete form

```
SCH NAME [Formals] IS [DeclPart | Expression] [ST Predicate] END
```

'SCH' , SchemaName, 'IS' ,DeclPart, 'ST',AxiomPart, 'END'

'SCH' , SchemaName, 'IS' ,DeclPart, 'END'

Ident, '==' ,Expression

Sample representation and transformation

Note: A Global Definition defines a new *schema.* There are two forms for a schema definition. The horizontal is the primary form. The vertical form, using a schema box, is given a meaning in terms of an equivalent horizontal definition.

Representation	Abstract

Type When a schema or variable is declared the name is added to the type-environment and is mapped to the type of the schema or expression.

$$(\!|\mathbf{abbr}\, N \mathrel{\widehat=} X |\!)^{\mathcal{T}} \;=\; \langle 1, \langle N^{\circ}, [\![X]\!]^{\mathcal{T}} \rangle \fatsemi \{-\} \rangle \fatsemi \oplus$$

Meaning When a schema or variable is declared the name of the schema is added to the environment and is mapped to the meaning of the schema or expression.

$$(\!|\mathbf{abbr}\, N \mathrel{\widehat=} X |\!)^{\mathcal{M}} \;=\; \langle 1, \langle N^{\circ}, [\![X]\!]^{\mathcal{M}} \rangle \fatsemi \{-\} \rangle \fatsemi \oplus$$

Note:

- The horizontal form of the definition defines the schema with name SchemaName as the schema denoted by the SchemaExpr.

- The vertical form of the definition defines the schema with name SchemaName as the schema denoted by the schema expression constructed from the schema text comprising the horizontal equivalents of the DeclPart and the AxiomPart (see Vertical Form).

A SchemaName may be used to define only one schema within a specification.

A Schema has global scope except within the text of its definition. Recursive schema definitions are not allowed. The scope of variables introduced in the DeclPart is local to the SchemaDef and includes the AxiomPart.

10.7 Generic definition

A generic definition of variables adds these variables to the environment and maps them to a function from all possible instantiations of their generic parameters to the values of the variables with these instantiations.

Abstract syntax

GENERICDEF = **abbr** NAME[NAME, NAME, ..., NAME] := EXP

Concrete form

```
NAME [Formals] DEFINE_EQUAL Expression
```

Sample representation and transformation

Representation	Abstract

Type

$$(\!|\mathbf{abbr}N[S_1,\ldots,Sm] \mathrel{\widehat=} X |\!)^{\mathcal{M}} =$$
$$\langle$$
$$1,$$
$$\lambda(\langle 1, \langle \langle S_1{}^\circ, Ptype^\circ \,;\ni\rangle,\ldots,\langle S_m{}^\circ, Ptype^\circ \,;\ni\rangle\rangle \,;\, \{\ldots\}))\,;\,{}^{\exists}(\langle\langle [\![S_1]\!]^{\mathcal{T}},\ldots,[\![S_m]\!]^{\mathcal{T}}\rangle,[\![X]\!]^{\mathcal{T}}\rangle)$$
$$\rangle\,;\oplus$$

Value

Note:

In a Generic Definition, the DeclPart declares the names of the generic variables whose types can be determined upon instantiation of the formal parameters.

An abbreviation definition can be used to define a possibly generic variable which is named by an identifier Abbrev.

The variable defined by the expression can take three forms:

- Possibly Generic Variable Ident.
- Prefix Generic Symbol PreGen.
- Infix Generic Symbol InGen.

In the latter two cases, the names of the generic parameters, Word indicate the positions of the actual parameters which can be supplied when the variables are used.

A schema may be defined with generic parameters and when used it must be always instantiated.

10.8 Conjecture

A new section – text to be added.

Abstract syntax

$$\mathsf{CONJECTURE} = \mathbf{conj}\ \mathsf{SCHEMA} \dagger \ldots \dagger \mathsf{SCHEMA} \mid \mathsf{PRED}, \ldots, \mathsf{PRED} \vdash \mathsf{PRED}, \ldots, \mathsf{PRED}$$

□

11 Specification

Notes on this section of the Z Standard

Section title: Specification
Source file: spc.tex
Section editor:
Original text by: Stephen Brien
Contributions by:
Most recent update: 30th June 1995
Formatted: 3rd July 1995

Editor's note:

This section has not been revised. It will be re-written when the current discussions on semantics, which affect this section and the section on Paragraph, have been completed.

11.1 Introduction

A specification is constructed from a sequence of paragraphs:

Abstract syntax

$$\mathsf{SPEC} = \mathsf{PAR}\ ,\ldots,\ \mathsf{PAR}$$

Sample representation and transformation

Production	Representation	Abstract
[Paragraph] , {Narrative , Paragraph} , [Narrative]	P_1 Narrative ... Narrative P_n	$[\![P_1]\!]^{PAR}$ and ... and $[\![P_n]\!]^{PAR}$

Type A specification is well-typed if the empty type environment is in the domain of the typing relation.

Meaning The meaning of a specification is the set of environments which are related to the empty environment by the paragraphs of the text. These are all the environments which are enrichments of the empty environment by the specification. A sequence of paragraphs can be composed together. They denote a relation between environments. This relation is the sequential composition of the relations denoted by the individual paragraphs.

$$zmnP_1 \textbf{and} \ldots \textbf{and} P_n \;=\; {}^{\wedge}(\{\!|P_1|\!\}^{\mathcal{M}} ; \ldots ; \{\!|P_n|\!\}^{\mathcal{M}})\varnothing$$

Note *A Z specification consists of a sequence of paragraphs separated by paragraph separators. These paragraph separators may include explanatory text. The global signature and property are constructed from the meanings of these paragraphs.*

A paragraph is either a definition or a constraint.

A definition introduces Basic types, schemas, or variables (named elements, sets tuples or bindings) together with constraints on them. The effect of a definition is to augment the global signature and to conjoin its constraint with the global property.

A constraint denotes a property on variables and schemas declared elsewhere. The effect of a constraint is to conjoin its property with the global property.

A specification is well typed if every term and predicate within the paragraphs is well typed.

□

A Abstract syntax – Normative Annex

Notes on this section of the Z Standard

Section title: Abstract syntax

Note: This version of the Abstract syntax is based on ZSRC Document z-159v2.tex, with amendments agreed at Meeting 23 of the Z Standards Panel on 27th September 1994.

Section editor: John Nicholls
Contributions by: (to be added)
Source file: absyn.tex
Most recent update: 29th May 1995 (minor update)
Formatted: 3rd July 1995

A.1 Introduction

Basis of definition. The abstract syntax is central to the definition of Z. It stands between the concrete representations of Z documents – as marks on paper and images on screens – and the abstract entities, semantic relations and semantic functions used for defining their meaning.

There are many possible ways of constructing an abstract syntax for Z, and the choice of the form given below is a matter of judgement, taking into account the somewhat conflicting aims of simplicty and economy of semantic definition, and the maintenance of a clear relationship with the concrete representation.

The abstract syntax has the following objectives:

to identify and separately name the distinct categories of the notation.

to simplify and unify the underlying concepts of the notation, putting like things with like, and reducing unnecessary duplication.

The syntax is presented as a set of production rules, in which each entity is defined in terms of its constituent parts. For each of the entities defined in the abstract syntax, there is a subsection in the main part of the Base Standard, defining its representation and meaning.

Metalanguage. The definition uses the following notation:

::=	definition symbol
|	disjunction symbol

Several definitions contain lists of entities, separated by commas or other separating characters. Where there may be an arbitrary number of entities in such a list, the following notation is used:

...	ellipsis, denoting a finite (possibly zero) number of occurrences of the preceding entity, together with appropriate separators

Terminal entities. The terminal entities of the definition are semantic entities, written in uppercase sans-serif font.

In addition, the syntax definitions contain operators, symbols and keywords similar to those used in the concrete syntax. These are written in this way to indicate the relationship of each abstract definition with the concrete form of the notation.

The relationship between the abstract and concrete forms of each entity is indicated in the entity definitions in the main body of the standard, under the headings "Representation and transformation".

Changes in this version. The following changes have been made to the Abstract Syntax since the version published in Version 1.0.

Structural changes:

the previously separate entity Schematext has been removed and merged with Schema.

a new rule has been introduced allowing an expression to be written wherever a schema (or what was previously called schematext) is allowed. Such an expression must be suitably typed; it should be noted that type information is not expressed in the Abstract Syntax.

the entity Compound Schema has been removed from the Abstract Syntax. The semantics of Compound Schema is defined in terms of Schema Conjunction.

the entities Schema Designator (SDES) and Generic Schema Designator (SGENDES) have been removed.

Changes in presentation:

a more uniform convention for naming syntactic entities has been introduced.

the order of presentation has been modified.

A.2 Specification

SPEC = PAR ,..., PAR

A.3 Paragraph

PAR = GIVENSETDEF
| GLOBALPRED
| GLOBALSCHEMA
| GENERICSCHEMA
| GLOBALDEF
| GENERICDEF
| CONJECTURE

GIVENSETDEF = **given** [NAME, NAME, ..., NAME]

GLOBALPRED = **where** PRED

GLOBALSCHEMA = **def** SCHEMA

GENERICSCHEMA = **gendef** [NAME, NAME, ..., NAME] **const** SCHEMA

GLOBALDEF = **abbr** NAME := EXP

GENERICDEF = **abbr** NAME[NAME, NAME, ..., NAME] := EXP

CONJECTURE = **conj** SCHEMA† ... †SCHEMA | PRED, ..., PRED ⊢ PRED, ..., PRED

A.4 Schema

SCHEMA = SDECL
| SCONSTRUCTION
| SNEGATION
| SDISJUNCTION
| SCONJUNCTION
| SIMPLICATION
| SEQUIVALENCE
| SPROJECTION
| SHIDING
| SUNIVQUANT
| SEXISTSQUANT
| SUNIQUEQUANT
| SRENAMING
| SCOMPOSITION
| SDECORATION
| SSUBSTITUTION
| EXPSCHEMA

SDECL = NAME, NAME, . . . , NAME : EXP

SCONSTRUCTION = SCHEMA | PRED

SNEGATION = ¬ SCHEMA

SDISJUNCTION = SCHEMA ∨ SCHEMA

SCONJUNCTION = SCHEMA ∧ SCHEMA

SIMPLICATION = SCHEMA ⇒ SCHEMA

SEQUIVALENCE = SCHEMA ⇔ SCHEMA

SPROJECTION = SCHEMA **Proj** SCHEMA

SHIDING = SCHEMA \ [NAME, . . . , NAME]

SUNIVQUANT = ∀ SCHEMA • SCHEMA

SEXISTSQUANT = ∃ SCHEMA • SCHEMA

SUNIQUEQUANT = $\exists_1$ SCHEMA • SCHEMA

SRENAMING = SCHEMA [NAME/NAME, . . . , NAME/NAME]

SCOMPOSITION = SCHEMA ⨟ SCHEMA

SDECORATION = SCHEMA DECOR

SSUBSTITUTION = EXP ⊙ SCHEMA

EXPSCHEMA = EXP

A.5 Predicate

$$
\begin{aligned}
\mathrm{PRED} &= \mathrm{EQUALITY} \\
&\mid \mathrm{MEMBERSHIP} \\
&\mid \mathrm{TRUTH} \\
&\mid \mathrm{FALSEHOOD} \\
&\mid \mathrm{NEGATION} \\
&\mid \mathrm{DISJUNCTION} \\
&\mid \mathrm{CONJUNCTION} \\
&\mid \mathrm{IMPLICATION} \\
&\mid \mathrm{EQUIVALENCE} \\
&\mid \mathrm{UNIVERSALQUANT} \\
&\mid \mathrm{EXISTSQUANT} \\
&\mid \mathrm{UNIQUEQUANT} \\
&\mid \mathrm{SPRED} \\
&\mid \mathrm{PREDSUBSTITUTION}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{EQUALITY} &= \mathrm{EXP} = \mathrm{EXP} \\
\mathrm{MEMBERSHIP} &= \mathrm{EXP} \in \mathrm{EXP} \\
\mathrm{TRUTH} &= \mathbf{true} \\
\mathrm{FALSEHOOD} &= \mathbf{false} \\
\mathrm{NEGATION} &= \neg\, \mathrm{PRED} \\
\mathrm{DISJUNCTION} &= \mathrm{PRED} \lor \mathrm{PRED} \\
\mathrm{CONJUNCTION} &= \mathrm{PRED} \land \mathrm{PRED} \\
\mathrm{IMPLICATION} &= \mathrm{PRED} \Rightarrow \mathrm{PRED} \\
\mathrm{EQUIVALENCE} &= \mathrm{PRED} \Leftrightarrow \mathrm{PRED} \\
\mathrm{UNIVERSALQUANT} &= \forall\, \mathrm{SCHEMA} \bullet \mathrm{PRED} \\
\mathrm{EXISTSQUANT} &= \exists\, \mathrm{SCHEMA} \bullet \mathrm{PRED} \\
\mathrm{UNIQUEQUANT} &= \exists_1\, \mathrm{SCHEMA} \bullet \mathrm{PRED} \\
\mathrm{SPRED} &= \mathrm{SCHEMA} \\
\mathrm{PREDSUBSTITUTION} &= \mathrm{EXP} \odot \mathrm{PRED}
\end{aligned}
$$

A.6 Expression

$$
\begin{aligned}
\mathrm{EXP} &= \mathrm{IDENT} \\
&\mid \mathrm{GENINST} \\
&\mid \mathrm{NUMBERL} \\
&\mid \mathrm{STRINGL} \\
&\mid \mathrm{SETEXTN} \\
&\mid \mathrm{SETCOMP} \\
&\mid \mathrm{POWERSET} \\
&\mid \mathrm{TUPLE} \\
&\mid \mathrm{PRODUCT} \\
&\mid \mathrm{TUPLESELECTION} \\
&\mid \mathrm{BINDINGEXTN} \\
&\mid \mathrm{THETAEXP} \\
&\mid \mathrm{SCHEMAEXP} \\
&\mid \mathrm{BINDSELECTION} \\
&\mid \mathrm{FUNCTAPP} \\
&\mid \mathrm{DEFNDESCR} \\
&\mid \mathrm{IFTHENELSE} \\
&\mid \mathrm{EXPSUBSTITUTION}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{IDENT} &= \mathrm{NAME} \\
\mathrm{GENINST} &= \mathrm{NAME}\ [\mathrm{EXP}, \mathrm{EXP}, \ldots, \mathrm{EXP}] \\
\mathrm{NUMBERL} &= \mathrm{NUMBER} \\
\mathrm{STRINGL} &= \mathrm{STRING} \\
\mathrm{SETEXTN} &= \{\mathrm{EXP}, \mathrm{EXP}, \ldots, \mathrm{EXP}\} \\
\mathrm{SETCOMP} &= \{\mathrm{SCHEMA} \bullet \mathrm{EXP}\} \\
\mathrm{POWERSET} &= \mathbf{Pow}\ \mathrm{EXP} \\
\mathrm{TUPLE} &= (\mathrm{EXP}, \mathrm{EXP}, \ldots, \mathrm{EXP}) \\
\mathrm{PRODUCT} &= \mathrm{EXP} \times \mathrm{EXP} \times \ldots \times \mathrm{EXP} \\
\mathrm{TUPLESELECTION} &= \mathrm{EXP}\ .\ \mathrm{NUMBERL} \\
\mathrm{BINDINGEXTN} &= (\!|\ \mathrm{NAME} := \mathrm{EXP}, \ldots, \mathrm{NAME} := \mathrm{EXP} |\!) \\
\mathrm{THETAEXP} &= \theta\ \mathrm{SCHEMA}\ \mathrm{DECOR} \\
\mathrm{SCHEMAEXP} &= \mathrm{SCHEMA} \\
\mathrm{BINDSELECTION} &= \mathrm{EXP}\ .\ \mathrm{NAME} \\
\mathrm{FUNCTAPP} &= \mathrm{EXP}\ (\ \mathrm{EXP}\ ) \\
\mathrm{DEFNDESCR} &= \mu\ \mathrm{SCHEMA} \bullet \mathrm{EXP} \\
\mathrm{IFTHENELSE} &= \mathbf{if}\ \mathrm{PRED}\ \mathbf{then}\ \mathrm{EXP}\ \mathbf{else}\ \mathrm{EXP}\ \mathbf{fi} \\
\mathrm{EXPSUBSTITUTION} &= \mathrm{EXP} \odot \mathrm{EXP}
\end{aligned}
$$

A.7 Name

NAME = WORD DECOR, ... , DECOR

□

B Concrete syntax – Normative Annex

Notes on this section of the Z Standard

Section title: Concrete syntax
Note: This version of the syntax is based on the proposal by Will Harwood and Pete Steggles (Document 173 dated 6th March 1995).
Section editor: John Nicholls
Contributions by: Will Harwood, Pete Steggles, Chris Sennett, Rob Arthan, Stephen Brien, ... (more to be added)
Source file: concrete.tex
Most recent update: 30th June 1995
Formatted: 3rd July 1995

B.1 Introduction

Editor's note: This Annex and the Lexis (Annex C), replace the section previously called Representation Syntax.

The relationships between the different forms of syntax, and the metalanguages used for their description, need further revision.

The concrete syntax and lexis are designed to meet the following requirements:

- to be as close as possible to 'traditional' Z;
- to permit the substitution of equals for equals;
- to make unparsing injective (i.e. different legal ASTs should have different unparsed forms) and total (i.e. there should be a concrete syntax for every legal AST);
- to make it convenient to project representations of Z and enter text by keyboard.

Editor's note: Further notes to be added here . . .

B.2 Syntactic metalanguage

The concrete syntax and lexis are defined using a BNF notation based on:

> BSI Standard **BS 6154**, *Method of defining syntactic metalanguage*, British Standards Institution, 1981.

The following symbols are used:

,	concatenate symbol
=	define symbol
|	definition separator symbol
[ ]	enclose optional syntactic items
{ }	enclose syntactic items which may occur zero or more times
' '	enclose terminal symbols
;	terminator symbol denoting the end of a rule
−	subtraction from a set of terminals
? ... ?	User defined rule

Precedence. The concatenate symbol has a higher precedence than the definition separator symbol.

Naming conventions. The following naming conventions are used:

- terminals are fully capitalized, e.g. `ELSE`.
- non-terminals are partly capitalized, e.g. `DeclPart`.

Editor's note: Discuss the use of `tt` typeface here.

Editor's note: The following comment is taken from D-173 and needs to be noted in future revisions:

The abstract syntax onto which this syntax is targetted is a slightly modified version of the one in ZSRC Document z-159. The changes are as follows:

- EXP can be an option of SCHEMA.
- The decoration is removed from THETAEXP.
- The SCHEMASUBSTITUTION, SDES, GENSDES options are removed.

B.3 Paragraph

```
Paragraph  =
             SQBRA NameList1 SQKET
           | Predicate
           | AX [DeclPart | Expression] [ST Predicate] END
           | GEN [Formals] BAR [DeclPart | Expression] [ST Predicate] END
           | SCH NAME [Formals] IS [DeclPart | Expression] [ST Predicate] END
           | NAME [Formals] DEFINE_EQUAL Expression
           | TURNSTILE Predicate
           | NAME FREEEQUALS Branch {VBAR Branch}
           | Fixity

Formals    = SQBRA NameList1 SQKET

Branch     = NAME [FREEBRA Expression FREEKET]
```

The top level paragraph syntax includes given set definitions, top level predicates, all the boxes, inline definitions, goals and operator template definitions.

The tokens: `AX`, `BAR`, `END`, `GEN`, `IS`, `SCH`, `ST` have special graphical conventions associated with them.

Informal text is treated as whitespace by the lexer.

Editor's note: The syntax for Specification has been (temporarily) omitted.

B.4 Fixity

> **Editor's note:** In later versions, the descriptions of templates and fixity may be moved to a different place.

```
Fixity      = SYNTAX Category Template
```

Operator definitions consist of a Category definition and Template definition.

```
Category    = REL
            | LEFT_FUN Precedence
            | RIGHT_FUN Precedence
```

The Category definition indicates whether the defined operator is a relation (`REL`), a left associative function (`LEFT_FUN`) or a right- associative function (`RIGHT_FUN`). Functions also have a numeric precedence defined.

```
Precedence      = NUMBER
```

The Precedence definition defines the precedence of the declared operator. There are at least 10000 precedence levels, numbered 0 to 9999. Higher numbers denote higher precedences.

```
Template    = [Arg] NAME {SeqArg NAME} [Arg]

Arg         = NORMAL
            | TYPE

SeqArg      = Arg
            | SEQUENCE BRA Expression COMMA Expression COMMA Expression KET
```

The `Template` definition is an alternating sequence of names and argument slots. There are three types of argument slots: `NORMAL`, which corresponds to the argument slots in current Z infix and prefix declarations; `TYPE`, which declares that the appropriate argument is also an actual generic parameter (so that a sequence of `TYPE` parameters corresponds to the same-order sequence of formal generic parameters of the operator being declared); `SEQUENCE`, which is an argument slot for a comma-separated list of expressions.

The effect of a syntactic template definition is:

- to assign suitable lexical status to the tokens which occur in the template.
- to assert the existence of a 'compound symbol' which represents the template.
- to assign an appropriate precedence and associativity to this compound symbol.
- to retain information about this symbol sufficient to identify it as a relation or function and to cope with its generic instantiation (if it has TYPE argument slots).

The following token categories are defined (where names postfixed with P represent the relational version of the token used in `Predicate`)

```
I, IP           infix token
PRE, PREP       prefix token
POST, POSTP     postfix token
L, LP           initial token
EL, ELP         initial token preceded by expression
ES              separator token preceded by expression
SS              separator token preceded by expression commalist
ER, ERP         final token preceded by expression
SR, SRP         final token preceded by expression commalist
ERE, EREP       final token preceded by expression and
                followed by expression
SRE, SREP       final token preceded by expression commalist and
                followed by expression
```

Any attempt to redefine the lexical status of a token is an error.

Here are some example templates with descriptions of their effects.

```
SYNTAX REL small NORMAL                                    (small => PREP)
SYNTAX REL NORMAL isodd                                    (isodd => POSTP)
SYNTAX 100 add NORMAL to TYPE                              (add => L; to => ERE)
SYNTAX 50  add SEQUENCE ({},makeSet,setUnion) to TYPE (add => L; to => SRE)
SYNTAX RIGHT 900 ARG a_normal_infix ARG                    (a_normal_infix => I)
```

The sequence argument slot includes a triple of a zero, unit injection, and 'union' function, which are used at parse-time to construct the appropriate expression from the list of parsed elements. We choose the triple including union because the three constituents are generally defined for most generic collections anyway.

B.5 Predicate

```
Predicate =
            Expression
          | Predicate CONJ Predicate
          | PRED Expression
          | EXISTS TextOrExpression DOT Predicate
          | EXISTS1 TextOrExpression DOT Predicate
          | FORALL TextOrExpression DOT Predicate
          | Predicate IFF Predicate
          | Predicate IMPLIES Predicate
          | Predicate OR Predicate
          | Predicate AND Predicate
          | NOT Predicate
          | Relation
          | Expression SUBST Predicate
          | BRA Predicate KET
          | TRUE
          | FALSE
```

Any ambiguity in the above grammar is resolved by alloting precedences to productions. The precedences of productions increase as we go down the page. All relevant operators are left-associative apart from `IMPLIES` and `SUBST` which are right-associative.

The operators should all be familiar except `CONJ`. This is a low-precedence conjunction operator which the lexer may return as the result of applying some layout rule.

B.5.1 Schemas as predicates

Because the schema expression connectives `AND`, `OR`, `NOT` etc. are lexically identical to the predicate connectives, we need a way of clarifying our intentions in ambiguous cases. The precedence rules above ensure that any expression involving these connectives will be parsed as a predicate if it can be; to force interpretation as a schema expression, we simply prefix an expression with the `PRED` coercion operator.

B.5.2 Relation application

```
Relation  = PrefixRel Expression
          | Expression PostfixRel
          | Expression InfixRel Expression {InfixRel Expression}
          | NofixRel
```

Relation applications are parsed as above, using the extended 'grammar for operators' which uses the tokens defined by the template mechanism.

B.5.3 Relations

```
PrefixRel  = LP {Expression ES | ExpressionList SS}
             (Expression EREP | ExpressionList SREP)
           | PREP

PostfixRel = ELP {Expression ES | ExpressionList SS}
             (Expression ERP | ExpressionList SRP)
           | POSTP

InfixRel   = ELP {Expression ES | ExpressionList SS}
             (Expression EREP | ExpressionList SREP)
           | IP
           | MEMBER
           | EQUALS

NofixRel   = LP {Expression ES | ExpressionList SS}
             (Expression ERP | ExpressionList SRP)
```

For a detailed explanation of the relation 'grammar for operators' refer to the appendix.

B.6 Expression and schema expression

```
Expression =
              IF Predicate THEN Expression ELSE Expression
            | EXISTS TextOrExpression DOT Expression
            | EXISTS1 TextOrExpression DOT Expression
            | FORALL TextOrExpression DOT Expression
            | MU TextOrExpression DOT Expression
            | LAMBDA TextOrExpression DOT Expression
            | Expression IFF Expression
            | Expression IMPLIES Expression
            | Expression OR Expression
            | Expression AND Expression
            | NOT Expression
            | Expression COMPOSE Expression
            | Expression HIDING BRA NameList1 KET
            | Expression PROJECTION Expression
            | Expression SUBST Expression
            | Expression CROSS {Expression CROSS} Expression
            | PSET Expression
            | Prefix Expression
            | Expression Postfix
            | Expression Infix Expression
            | Expression Expression {Expression}
            | Expression DECORATION
            | Expression SQBRA RenameList SQKET
            | Expression SELECT NAME
            | Expression SELECT NUMBER
            | THETA Expression
            | Nofix
            | NAME SQBRA ExpressionList1 SQKET
            | SETBRA ExpressionList SETKET
            | SETBRA TextOrExpression DOT Expression SETKET
            | SETBRA Text SETKET
            | BINDERBRA BindList BINDERKET
            | BRA Expression COMMA ExpressionList1 KET
            | BRA MU TextOrExpression KET
            | BRA Expression KET
            | STRING
            | NUMBER
            | SQBRA Text SQKET
```

Again, precedences increase as we go down the page, and all operators are left-associative except `IMPLIES` and `SUBST` which are right-associative.

In Standard Z schemas are expressions and expressions are schemas. There are two advantages to this:

- in proof, it upholds the substitutivity of equals for equals.

- in specification, it allows perfectly sensible idioms which are currently banned (for example, if I have a function which returns sets of some binding, why can't I use the result of an application of the function in a normal schema expression?).

We can decide whether something should be evaluated in a 'declaration' way or in a 'set of bindings' way by looking at where it occurs. (We presume that for all x,y (eval-as-set-of-bindings x) = (eval-as-set-of-bindings y) iff (eval-as-declaration x) = (eval-as-declaration y)).

Notice that now we can decorate an arbitrary expression and rename an arbitrary expression (although these only make sense in type terms when the expression involves bindings).

Notice that mu expressions with no dot are bracketed because the schema text objects inside are very low precedence.

Templates use an extended 'grammar for operators' similar to the one used by relations:

```
Prefix  = L {Expression ES | ExpressionList SS}
          (Expression ERE | ExpressionList SRE)
        | PRE

Postfix = EL {Expression ES | ExpressionList SS}
          (Expression ER | ExpressionList SR)
        | POST

Infix   = EL {Expression ES | ExpressionList SS}
          (Expression ERE | ExpressionList SRE)
        | I

Nofix   = L {Expression ES | ExpressionList SS}
          (Expression ER | ExpressionList SR)
        | NAME
```

The template application is similar to that in the predicate section, but here we obviously don't have the same kind of chaining; when a chain of templates is parsed the resulting syntax tree must be rearranged to a form consistent with the precedence and associativity figures using a leftmost derivation using an algorithm such as [16]. Thus if we have a juxtaposition of a right associative operator $\_p\_$ and a left associative operator $\_q\_$ of equal precedence, the formula $x\ p\ y\ q\ z$ parses as $(x\ p\ y)\ q\ z$.

Note that the precedences defined on templates only work with respect to other templates – other schemes are hard to process for user and machine.

The traditional grammar for set comprehensions and displays has an ambiguity – the distinction of comprehensions with no `DOT Expression` part from one-element displays; the traditional 'solution' would be to put brackets in like this: `{(Expression)}`; this is a crime against brackets. Our proposal has no ambiguity, and relies on the distinction between a Schema Text and a Schema Expression; given any Schema Expression `S`, `{S}` is a display and `{S|true}` is a comprehension.

B.6.1 Schema texts

```
TextOrExpression =
              Text
            | Expression

Text        = DeclPart [VBAR Predicate]

DeclPart    = DeclElem SEMICOLON DeclElem {SEMICOLON DeclElem}
            | BasicDecl

DeclElem    = BasicDecl
            | Expression

BasicDecl   = NameList1 COLON Expression
```

There is no explicit abstract syntax for schema texts but it is important to still have things which look like schema texts at the level of concrete syntax.

We use a grammar for inline schema texts which can have any schema in a declaring position. Notice that for an arbitrary schema expression S the inline schema text [S] is not allowed – instead the user should write simply S.

B.6.2 Binding and renaming

```
Bind        = NAME DEFINE_EQUAL Expression
Rename      = NAME RENAME NAME
```

There should be no surprises here – this is exactly the same as the traditional Z grammar.

B.7 Lists

```
ExpressionList    = [ExpressionList1]
ExpressionList1   = Expression {COMMA Expression}

NameList    = [NameList1]
NameList1   = NAME {COMMA NAME}

BindList    = [BindList1]
BindList1   = Bind {COMMA Bind}

RenameList = Rename {COMMA Rename}
```

B.8 Interface to the lexical analyzer

B.8.1 Layout rules

Layout information is used in traditional Z specifications

- to replace the semicolons in the declaration parts of vertical boxes.
- to replace the conjunctions in the predicate parts of vertical boxes.

The parser relies on the lexer to find out when layout information is being used and to insert the appropriate separators (and brackets in the case of predicate conjunctions) into the token stream (the traditional approach to dealing with layout questions is to resolve them at the lexical analysis stage (cf Occam, Haskell)).

B.8.2 Decorations, words and names

The parser relies on the lexer to recognise `DECORATION` and `NAME`. Note that the parser has no idea whether a name has any decorations in it; there is no notion of Word in the abstract syntax (and hence in the parser).

Note that some care needs to be shown to distinguish between the decorated expression

```
(decorate (name "f") "´")
```

and the name

```
(name "f´")
```

With a normal treatment of whitespace, the string `"f ´"` would parse as the former and the string `"f´"` would parse as the latter.

B.9 Operator definition using templates

B.9.1 Prefix, Postfix, Infix, Nofix

A token f corresponding to a Word in traditional Z can have one of four lexical categories:

Nofix tokens just correspond to ordinary Words.

Prefix tokens must be followed by an expression. The prefix token f has a corresponding nofix token $f\_$.

Postfix tokens must be preceded by an expression. The postfix token f has a corresponding nofix token $\_f$.

Infix tokens must be followed and preceded by expressions. The infix token f has a corresponding nofix token $\_f\_$.

Prefix, postfix and infix tokens can be declared by declaring the corresponding nofix tokens (though normally only at top-level in support tools because parsing would otherwise require typechecker services). This provides a simple mechanism for coping with the common mathematical operators, allowing the abstract syntax to ignore fixity issues by simply using the appropriate nofix tokens.

We can define a grammar for this kind of operator definition:

```
Fixity = [_] Word [_]
```

(where square brackets denote an optional expression). In a conventional implementation of a parser for Z fixity definitions will pass appropriate token information to the lexical analyser. For example, the statement

```
_ f
```

would define f as a postfix token and $\_f$ as a nofix token. The parser would then be able to parse a postfix application

```
x f
```

generating the abstract syntax

```
(functapp (name "_f") x)
```

so all fixity issues are a matter for the lexer and the parser, and may be ignored at the level of abstract syntax.

But this regime is not sufficiently powerful to handle even basic Z toolkit operators such as relational image, which is of the form $\_f_1\_f_2$. Traditionally, specifiers have got round these problems by defining combinations of tokens; for example, $\_f_1\_f_2$ can be mimicked by defining two tokens $\_f_1\_$, and $\_f_2$; this is not a nice solution because the correspondence with a single nofix token is lost and so the abstract syntax contains excess structure.

B.9.2 Adding structure to token bodies

To get round this problem we can consider adding internal structure to the (ex-) token f so that f now corresponds to the structure $f_1\_f_2$; f_1 and f_2 are a pair of tokens which must stand either side of an expression.

The new operator definition syntax is then

```
Fixity = [_] Word {_ Word} [_]
```

(where curly brackets denote zero or more occurrences of an expression). We could then define a token like relational image thus:

```
_ f1 _ f2
```

which would define f_1 and f_2 suitably, and $\_f_1\_f_2$ as a nofix token.

So how can f_1 and f_2 be suitably defined, such that we can generate a grammar for the more complex operator applications?

B.9.3 Parsing structured tokens

Our approach is to define an additional set of token types. As well as `INFIX`, `PREFIX`, `POSTFIX` and `NAME` we now add

L tokens at the start of a 'structured token' which are not preceded by an expression.

EL tokens at the start of a 'structured token' which are preceded by an expression.

S tokens inside a 'structured token'.

R tokens at the end of a 'structured token' which are not followed by an expression.

RE tokens at the end of a 'structured token' which are followed by an expression.

and provide a 'grammar for operators' thus:

```
Prefix  = L {Expression S} Expression RE
        | PREFIX

Postfix = EL {Expression S} Expression R
        | POSTFIX

Infix   = EL {Expression S} Expression RE
        | INFIX

Nofix   = L {Expression S} Expression R
        | NAME
```

Some section of the Z grammar which previously looked like this:

```
Expression = PREFIX Expression
    | Expression POSTFIX
    | Expression INFIX Expression
    | NAME
```

would now look like this:

```
Expression = Prefix Expression
    | Expression Postfix
    | Expression Infix Expression
    | Nofix
```

This gives us a unified mechanism for defining operators which is powerful enough to define the basic forms of all the toolkit operators. In order to handle more subtle conditions we need to add a few complications to the basic scheme.

B.10 Generics

In Z there is a notion of 'infix generics'. For example, the function arrow operator is a function which uses its arguments as generic instantiations:

```
X -> Y
```

actually equals

```
(->[X,Y])(X,Y)
```

We could add another kind of argument slot, #, for an argument which is used as a generic instantiation, so that we could write the template:

```
# -> #
```

This has the advantage that we can also say things like:

```
add _ to #

add x to xs = (add_to_ [xs])(x,xs)
```

where the above operator puts items into sets.

B.11 Precedence and associativity

In order to resolve ambiguities, we need to use precedence and associativity information. The approach adopted in the standard is to supply one numeric precedence value and a choice of left or right associativity.

B.11.1 Relations

Relations can be defined using the same kinds of mechanism as functions, but their instances are interpreted as set membership statements rather than function applications.

To provide an analogous grammar for relational operators, we use the same scheme as for functional operators, with minor modifications.

L tokens become LP tokens.

EL tokens become LP tokens.

S tokens stay the same.

R tokens become LP tokens.

RE tokens become LP tokens.

and the grammar for operators becomes:

```
PrefixRel  = LP {Expression S} Expression REP
           | PREFIXREL

PostfixRel = ELP {Expression S} Expression RP
           | POSTFIXREL

InfixRel   = ELP {Expression S} Expression REP
           | INFIXREL

NofixRel   = LP {Expression S} Expression RP
```

The basic grammar for Relation application would look like this:

```
Relation = PrefixRel Expression
         | Expression PostfixRel
         | Expression InfixRel Expression
         | NofixRel
```

Most versions of Z allow iterated infix relations:

```
Relation = PrefixRel Expression
         | Expression PostfixRel
         | Expression InfixRel Expression {InfixRel Expression}
         | NofixRel
```

We could, of course, go further down this road, giving us:

```
Relation = ([PrefixRel] Expression {InfixRel Expression} [PostfixRel])
           - Expression
         | NofixRel
```

(where 'A - B' denotes subtraction of the production set B from A) but popular opinion seems to consider this a step too far.

B.11.2 Sequences

When an expression is enclosed on both sides by tokens, we have an opportunity; we could also permit a comma-separated list of expressions to occur in this situation. This is useful in defining display operators.

To do this, we provide more tokens:

S splits into ES and SS.

R splits into ER and SR.

RE splits into ERE and SRE.

RP splits into ERP and SRP.

REP splits into EREP and SREP.

The prefixed E indicates that a single expression is expected before the token; while the prefixed S indicates that a sequence of expressions (separated by commas) is expected before the token.

The grammars for operators change in the obvious way, giving us the final scheme which is used in the Z Standard syntax definition.

In the template definition syntax, the sequence argument slot includes a triple of a zero, unit injection, and 'union' function, which are used at parse-time to construct the appropriate expression from the list of parsed elements. We choose the triple including union because the three constituents are generally defined for most generic collections anyway. Here is how one might go about defining the syntax of a sequence display operator (where the constituents of the triples have the obvious meanings).

```
fixity lseq ... (emptyList,makeSingletonSequence,concatenate) rseq
```

So that

```
lseq x,y,z rimg
```

parses as

```
(functapp (name "lseq_rseq")
   (functapp (name "concatenate")
      (functapp (name "concatenate")
         (functapp (name "makeSingletonSequence") (name "x"))
         (functapp (name "makeSingletonSequence") (name "y")))
      (functapp (name "makeSingletonSequence") (name "z"))))
```

and

```
lseq rimg
```

parses as

```
(functapp (name "lseq_rseq") (name "emptyList"))
```

□

C Lexis – Normative Annex

Notes on this section of the Z Standard

Section title: Lexis
Note: Based on D-167 v2 and D-177, with comments from syntax sub-committee meeting 18 May 1995; also comments from Meeting 27 of the Z Standards Panel.
Section editor: Susan Stepney (this version re-edited by JEN)
Contributions by: Chris Sennett, Rob Arthan, Trevor King, ... (more to be added)
Source file: lexis.tex
Most recent update: 29th June 1995
Formatted: 3rd July 1995

Editor's note: This Annex, though it has been revised, has not yet been fully updated to bring it into line with the toolkit Annex. The revision will be completed in the next version. JEN

C.1 Introduction

The Concrete Syntax (Annex B), uses named tokens to define its terminal symbols. This section defines the lexical grammar of those tokens in terms of Z *glyphs* (defined in section C.4). This section describes a typical rendering of the tokens, showing how they might be displayed on a printed page or a graphics screen. The detailed appearance of the tokens is device-dependent. A character-based, machine-representable format, and the Interchange Format representation based on SGML are defined in sections C.4.3 and Annex D.

C.2 Soft newlines

Most white space is not recognized by a lexical analyzer, but is used as a *separator* when recognising tokens. In two special contexts some white space (called a 'hard new line') is recognized: as a `SEMICOLON` in a `DeclPart`, as a `CONJ` in a `Pred`. The rule for distinguishing 'soft' (white space) and 'hard' (recognised) newlines in these contexts is given in the following rules.

1. Tokens that can appear in these contexts are assigned to a 'soft newline category', based on whether the token could start or end a declaration or predicate.

   - BOTH: new lines are soft before and after the token, because it can neither start nor end a declaration or predicate. (It is 'infix', for example, ':')
   - AFTER: new lines are soft after the token, because it cannot end a declaration or predicate. (It is 'prefix', for example, '[')

- BEFORE: new lines are soft before the token, because it cannot start a declaration or predicate. (It is 'postfix', for example, ']')
- NEITHER: no new lines are soft, because such a token could start or end a declaration or predicate. (It is 'nofix', for example, '*true*')

2. When a new line appears between two tokens in the relevant context, the newline categories of both are examined. If either allows the newline to be soft in that position, it is soft, otherwise it is hard (and hence recognised).

3. All newlines are soft outside a `DeclPart` or a `Pred`. So tokens that cannot appear in these contexts can be considered to be in category BOTH.

The `Fixity` paragraph allows the definition of various mixfix names, which are placed in the appropriate newline category (see section C.3.3). Other (ordinary) user declared names are 'nofix', and so are placed in NEITHER.

C.3 Tokens

```
ZToken = SIMPLE
       | BOX
       | MIXFIX
       | DECORATION
       | NAME
       | NUMBER
       ;
```

C.3.1 Simple tokens

```
SIMPLE = AND | ... | VBAR;
```

A typical rendering of these literal tokens is given below.

The third column defines the soft newline category of those tokens that can appear in the context of a DeclPart or Pred. The fourth column notes the representation syntax productions where they occur.

Token	Representation	Newline	Production
AND	$\wedge$	BOTH	Exp, Pred
BRA	(	AFTER	Exp, Pred, SeqArg
COLON	:	BOTH	BasicDecl
COMMA	,	BOTH	Exp, SeqArg, XList1
COMPOSE	;	BOTH	Exp
CONJ	(newline)		Pred
CROSS	$\times$	BOTH	Product
DEFINEEQUAL	==	BOTH	Bind, Paragraph
DOT	$\bullet$	BOTH	Exp
EQUALS	=	BOTH	Relation

ELSE	**else**	BOTH	Exp
EXISTS	$\exists$	AFTER	Exp, Pred
EXISTS1	$\exists_1$	AFTER	Exp, Pred
FALSE	*false*	NEITHER	Pred
FIXITY	*fixity*		Fixity
FORALL	$\forall$	AFTER	Exp, Pred
FREEBRA	$(\!|$		Branch
FREEEQUALS	::=		FreeType
FREEKET	$|\!)$		Branch
HIDING	$\setminus$	BOTH	Exp
IF	**if**	AFTER	Exp
IFF	$\Leftrightarrow$	BOTH	Exp, Pred
IMPLIES	$\Rightarrow$	BOTH	Exp, Pred
KET	)	BEFORE	Exp, Pred, SeqArg
LAMBDA	λ	AFTER	Exp
LEFTFUN	**leftfun**		Category
LET	**let**		Exp, Pred
MEMBER	$\in$	BOTH	Relation
MU	μ	AFTER	Exp
NORMAL	_		Arg
NOT	$\neg$	AFTER	Exp, Pred
OR	$\vee$	BOTH	Exp, Pred
PRED	**pred**	AFTER	Pred
PRESCH	pre	AFTER	Pred
PROJECTION	$\upharpoonright$	BOTH	Exp
PSET	$\mathbb{P}$	AFTER	Exp
REL	**rel**		Category
RENAME	/	BOTH	Rename
RIGHTFUN	**rightfun**	??	Category
SEMICOLON	;	BOTH	DeclElem, DeclPart
SEQUENCE	...		SeqArg
SELECT	.	BOTH	Exp
SETBRA	{	AFTER	Exp
SETKET	}	BEFORE	Exp
SQBRA	[	AFTER	Exp, Formals, Paragraph
SQKET	]	BEFORE	Exp, Formals, Paragraph
THEN	**then**	BOTH	Exp
THETA	θ	AFTER	Exp
TRUE	*true*	NEITHER	Pred
TURNSTILE	$\vdash$	??	Paragraph
TYPE	$	??	Arg
VBAR	$\mid$	BOTH	Paragraph, Text

C.3.2 Box tokens

```
BOX = AX | SCH | GEN      AFTER
    | IS | ST | BAR       BOTH
    | END                 BEFORE
    ;
```

A typical rendering of these BOX tokens is lines drawn around the Z text.

(Editor's note: add three examples.)

The Interchange Format (Annex E) defines a textual form.

C.3.3 Mixfix token categories

```
MIXFIX = I | ... | SREP
```

For the base language, these token categories are empty. They are populated by definitions using the template structure of the fixity paragraph, such as toolkit definitions.

The second column defines the soft newline category of names declared in these token categories. The third column notes the representation syntax productions where these tokens occur.

Token	Newline	Production
I, IP	BOTH	Infix
POST, POSTP	BEFORE	Postfix
PRE, PREP	AFTER	Prefix
EL, ELP	BOTH	Postfix, Infix
ER, ERP	BEFORE	Postfix, Nofix
ERE, EREP	BOTH	Prefix, Infix
ES	BOTH	Prefix, Postfix, Infix, Nofix
L, LP	AFTER	Prefix, Nofix
SR, SRP	BEFORE	Postfix, Nofix
SS	BOTH	Prefix, Postfix Infix
SRE, SREP	BOTH	Prefix, Infix

C.3.4 Decoration, name and number tokens

```
DECORATION = STROKE, {STROKE};
NAME       = WORD, {STROKE};
NUMBER     = '0' | DIGIT1, {DIGIT};
```

All these tokens are in the soft newline category NEITHER.

Notice that the lexis allows a NAME to include STROKEs, and that the concrete syntax allows an expression to be decorated with STROKEs. When the expression is a NAME, the two cases are disambiguated by

white space: $x!$ is the undecorated NAME consisting of the WORD 'x' followed by the STROKE '!'; $x\,!$ is the decorated expression consisting of the NAME 'x' decorated with the STROKE '!'; $x!!$ is the decorated expression consisting of the WORD '$x!$' decorated with the STROKE '!'.

```
STROKE    = '′' | '!' | '?' | SUBSCRIPT;
SUBSCRIPT = DOWN, GLYPH, {DOWN, GLYPH}, UP
```

The DOWN and UP subscript delimiter tokens could be presented as in-line literals, or they could indicate a lowering/raising of the text, and possible size change. Such rendering details are not defined here.

```
ALPHASTRING  = {LETTER | DIGIT}
SYMBOLSTRING = {SYMBOL}
WORDPART     = '_', (ALPHASTRING | SYMBOLSTRING)

WORD = WORDPART, {WORDPART}
     | LETTER, ALPHASTRING, WORDPART
     | SYMBOL, SYMBOLSTRING, {WORDPART}
```

C.4 Glyphs

```
GLYPH = DIGIT | LETTER | SYMBOL | SPECIAL
```

The glyph sets forming GLYPH are disjoint. SYMBOL is a user-extensible glyph set for making new symbolic identifiers: this is where characters from other alphabets (such as Japanese or Russian) can be added.

```
DIGIT1  = '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
DIGIT   = '0' | DIGIT1;
LETTER  = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm'
        | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'
        | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M'
        | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'
        | 'Γ' | 'Δ' | 'Θ' | 'Λ' | 'Ξ' | 'Π' | 'Σ' | 'Υ' | 'Φ' | 'Ψ' | 'Ω'
        | 'α' | 'β' | 'γ' | 'δ' | 'ϵ' | 'ζ' | 'η' | 'θ' | 'ι' | 'κ' | 'λ' | 'μ'
        | 'ν' | 'ξ' | 'π' | 'ρ' | 'σ' | 'τ' | 'υ' | 'ϕ' | 'χ' | 'ψ' | 'ω'
        ;
SYMBOL  = '$' | ',' | '.' | '/' | ':' | ';' | '=' | '[' | ']' | '{' | '|' | '}'
        | '∧' | '∨' | '¬' | '⇒' | '⇔' | '∀' | '∃'
        | '×' | 'ℙ' | '•' | '⦇' | '⦈' | '⊢'
        | '℘' | '↾' | '\'
        | any toolkit glyphs, including MIXFIX (as supported)
        | any user glyphs, including MIXFIX
        ;
SPECIAL = BOX
        | '′' | '!' | '?'
        | '(' | ')' | '_'
        ;
```

C.4.1 Examples

A glyph in the LETTER or DIGIT class may be rendered differently for reasons of emphasis or aesthetics, but it still represents the same glyph. For example, 'd', 'd', '**d**' and '**d**' are all the same glyph. This explains why those Greek characters that are identical in appearance to Roman characters do not appear twice in the list of letters.

A glyph in the SYMBOL class, which includes user-defined glyphs, must appear the same wherever it occurs in a specification. For example, schema composition $\fatsemi$ and the toolkit glyph relational composition $;$ are different glyphs.

Once a NAME has been recognised, if it consists of the glyph string corresponding to a *SIMPLE* token, it is recognised as that token instead. For example, $\mathbb{P}$ is recognised as *PSET*, but $\mathbb{P}_1$ ('$\mathbb{P}$', DOWN, '1', UP) and $\mathbb{P}_{45}$ ('$\mathbb{P}$', DOWN, '4', UP, DOWN, '5', UP) are recognised as NAMEs. For example, $\exists_1$ ('$\exists$', DOWN, '1', UP) is recognised as *EXISTS*1, but $\exists_0$ ('$\exists$', DOWN, '0', UP) is recognised as a NAME.

Underscore, '_', is used in words to separate strings of alphanumerics from strings of symbols. For example, '$x\_\diamond\diamond\_y$' is a single word, whereas '$x \diamond \diamond y$' consists of the three words 'x', '$\diamond\diamond$' and 'y'. For example, '$\mathbb{P}\_X$' is a single word, whereas '$\mathbb{P}\, X$' is the token *PSET* followed by the word 'X'.

(Editor's note: more examples here?)

C.4.2 Glyph representations

We present the glyphs of the Base language in a variety of formats, designed for different purposes.

Spoken name A suggested form for reading the glyph out loud, designed for use in reviews, or for discussing specifications over the telephone. (An English language form only is given; other natural languages may well use other forms.)

Interchange The format for use with the SGML interchange format (chapter ????)

Email The format for rendering the glyph on a low resolution device, such as a character-based terminal, or e-mail conversation. (The email form for digits and the Roman alphabet is the obvious one, and is not given explitily.)

The character **%** is used to flag a special string, for example $\times$ as **%x**, and disambiguate it from, for example the name **x**, to ease machine processing. This flag character may be omitted to reduce clutter, if there is no intention to machine-process the text.

Mathematical The format for rendering the glyph on a high resolution device, such as a bit-mapped screen, or on paper (either hand-written, or printed).

Other formats may be used for other purposes, as required.

Toolkit glyphs are described in the toolkit chapter.

C.4.3 Base language glyphs

Spoken name	Interchange	Email	Mathematical
and	&and	/\	$\wedge$
left [parenthesis]	(	(	(
colon	:	:	:
comma	,	,	,
schema compose	&scomp	%%;	$\fatsemi$
cross	&times	%x	$\times$
define equal	==	==	==
fat dot | spot	&bull	@	$\bullet$
equals	=	=	=
else	else	**else**	**else**
exists	&exist	%E	$\exists$
unique exists	&exist1	%E1	$\exists_1$
false	false	**false**	*false*
fixity	fixity	**fixity**	*fixity*
for all	&forall	%A	$\forall$
left chevron [bracket]	&lchev	<<	$\langle\!\langle$
free equals	::=	::=	::=
right chevron [bracket]	&rchev	>>	$\rangle\!\rangle$
hide	&hide	%\	$\backslash$
if	if	**if**	**if**
equivalent | if and only if	&iff	<=>	$\Leftrightarrow$
implies	&rArr	==>	$\Rightarrow$
right [parenthesis]	)	)	)
left function	leftfun	**leftfun**	**leftfun**
let	let	**let**	**let**
member | in	&isin	%e	$\in$
not	&not	-;	$\neg$
argument	_	_	_
or	&or	\/	$\vee$
coerce predicate	&pred	**pred**	**pred**
pre[condition]	&pre	**pre**	pre
project	&proj	%| \	$\upharpoonright$
power [set]	&pset	%P	$\mathbb{P}$
relation	rel	**rel**	**rel**
rename	/	/	/
right function	rightfun	**rightfun**	**rightfun**
semi[colon]	;	;	;
sequence argument	...	...	...
select | dot	.	.	.
left set [bracket]	&lcub	{	{
right set [bracket]	&rcub	}	}
left square [bracket]	&lsqb	[	[
right square [bracket]	&rsqb	]	]

then	then	**then**	**then**
true	true	**true**	*true*
turnstile	&vdash	|-	⊢
type argument	&num	$	$
bar	&verbar	|	|

C.4.4 Greek alphabet glyphs

Spoken name	Interchange	Email	Mathematical
alpha	&alpha	%alpha	α
beta	&beta	%beta	β
gamma	&gamma	%gamma	γ
delta	&delta	%delta	δ
epsilon	&epsi	%epsilon	ϵ
zeta	&zeta	%zeta	ζ
eta	&eta	%eta	η
theta	&thetas	%theta	θ
iota	&iota	%iota	ι
kappa	&kappa	%kappa	κ
lambda	&lambda	%lambda	λ
mu	&mu	%mu	μ
nu	&nu	%nu	ν
xi	&xi	%xi	ξ
pi	&pi	%pi	π
rho	&rho	%rho	ρ
sigma	&sigma	%sigma	σ
tau	&tau	%tau	τ
upsilon	&upsi	%upsilon	υ
phi	&phis	%phi	ϕ
chi	&chi	%chi	χ
psi	&psi	%psi	ψ
omega	&omega	%omega	ω
big delta	&Delta	%Delta	Δ
big gamma	&Gamma	%Gamma	Γ
big theta	&Theta	%Theta	Θ
big lambda	&Lambda	%Lambda	Λ
big xi	&Xi	%Xi	Ξ
big pi	&Pi	%Pi	Π
big sigma	&Sigma	%Sigma	Σ
big upsilon	&Upsi	%Upsilon	Υ
big phi	&Phi	%Phi	Φ
big psi	&Psi	%Psi	Ψ
big omega	&Omega	%Omega	Ω

C.5 Toolkit glyphs

Glyph representations are provided for the new glyphs introduced in the Toolkit.

> **Editor's note:** The precise contents of this list depends on the chosen Tookit, and hence is subject to change.

Spoken name	Interchange	Email	Mathematical
not equal	&ne	/=	$\neq$
non in	&notin	%/e	$\notin$
empty [set]	&empty	(/)	$\varnothing$
subset	&sube	%c_	$\subseteq$
not subset	&nsube	%/c_	$\not\subseteq$
proper subset	&sub	%c	$\subset$
not proper subset	&nsub	%/c	$\not\subset$
[set] union	&cup	%u	$\cup$
[set] intersection	&cap	%n	$\cap$
set difference	&sdiff	\	$\setminus$
set symmetric difference	&ssdiff	\\	$\setminus\!\!\setminus$
generalised union	&Bigcup	%uu	$\bigcup$
generalised intersection	&Bigcap	%nn	$\bigcap$
finite sets	&fset	%F	$\mathbb{F}$
relation	&rel	<-->	$\leftrightarrow$
maplet	&map	|-->	$\mapsto$
[relational] compose	&rcomp	%;	$\fatsemi$
functional compose	&compfn	%o	$\circ$
domain restrict	&dres	<:	$\lhd$
range restrict	&rres	:>	$\rhd$
domain subtract	&dsub	<-:	$\mathbin{\lhd\!\!\!-}$
range subtract	&rsub	:->	$\mathbin{-\!\!\!\rhd}$
inverse	&tilde	~	$\sim$
left relational image bracket	&limg	(|	$(\!\mid$
right relational image bracket	&rimg	|)	$\mid\!)$
transitive closure | plus	&tcl	|+	$^+$
reflexive transitive closure | star	&rtcl	|*	*
[relational] override	&oplus	(+)	$\oplus$
[partial] function	&pfun	-|->	$\pfun$
total function	&rarr	-->	$\fun$
[partial] injection	&pinj	>-|->	$\pinj$
total injection	&rarrtl	>-->	$\inj$

[partial] surjection	&psur	-|->>	$\nrightarrow\!\!\!\!\rightarrow$
total surjection	&Rarr	-->>	$\rightarrow\!\!\!\!\rightarrow$
bijection	&bij	>-->>	$\rightarrowtail\!\!\!\!\rightarrow$
finite function	&fpfun	-||->	$\nrightarrow\!\!\!\!\!\rightarrow$
finite injection	&fpinj	>-||->	$\rightarrowtail\!\!\!\!\!\rightarrow$
nat[ural number]	&nat	%N	$\mathbb{N}$
integer	&int	%Z	$\mathbb{Z}$
rational	&rat	%Q	$\mathbb{Q}$
real [number]	&real	%R	$\mathbb{R}$
less than	&lt	<	$<$
less than or equal to	&le	<=	$\leq$
greater than	&gt	>	$>$
greater than or equal to	&ge	>=	$\geq$
up to	&nldr	..	..
plus	+	+	+
[unary] minus	&uminus	-	$-$
[binary] minus	&bminus	-	$-$
times	*	*	*
cardinality | hash	&num	#	#
[real] divide	&divide	%/	div
left seq[uence bracket]	&lang	%<	$\langle$
right seq[uence bracket]	&rang	%>	$\rangle$
filter	&filter	| \	$\restriction$
co-filter	&cofilter	| /	$\downarrow$
extract	&extract	/ |	$\upharpoonright$
co-extract	&coextract	\ |	$\downharpoonright$
concatenate	&frown	^	$\frown$
left bag [bracket]	&lbag	[|	$[\![$
right bag [bracket]	&rbag	|]	$]\!]$

□

D Mathematical toolkit – Normative Annex

Notes on this section of the Z Standard

Section title: Mathematical toolkit
Section editor: John Wordsworth
Contributions by: John Wordsworth, ... *(others to be added)*
Source file: toolkit.tex
Most recent update: 30th June 1995
Formatted: 3rd July 1995

Editor's note: In this version, the only major change to this Annex has been the addition of *fixity* definitions to some of the definitions.

Introduction

This section defines a Mathematical Toolkit or Library for use with the Z notation. The principle is that those constructions that can be defined in terms of others are included in the Toolkit rather than in the core notation—this simplifies the core notation.

Most users will want to make use of the constructions defined in this section. This can therefore be regarded as a *basic* Toolkit, which users may augment with their own definitions, or replace if these definitions are not suitable for their use.

In this version of the Z Standard, the list of defined items follows the customary list of Toolkit items. Later versions of the Standard may include further definitions and explanations, and will link the Toolkit to related work on the semantics and proof system for Z.

Definitions of the Mathematical Toolkit are informally explained and illustrated. In some cases an illustration for one part of the Toolkit may rely on terms defined earlier in the toolkit. Many of the definitions given here are generic with respect to one or more sets.

Editor's note: The following note appeared an an earlier version. It is retained for possible use.

> Instantiation of a generic definition can be performed with any appropriate sets, not necessarily the maximal sets of their types. However the informal descriptions of these definitions are often here expressed as if the sets used for instantiation were in fact types, since that is the way in which these definitions are commonly instantiated in Z specifications.
>
> Reviewers of the draft standard are invited to comment on this approach.

D.1 Sets

Name

$\neq$ – Inequality

$\notin$ – Non-membership

Definition

fixity **rel** $\_ \neq \_$
fixity **rel** $\_ \notin \_$

$$
\begin{array}{|l}
[X] \\
\hline
\_ \neq \_ : X \leftrightarrow X \\
\_ \notin \_ : X \leftrightarrow \mathbb{P}\, X \\
\hline
\forall\, x, y : X \bullet x \neq y \Leftrightarrow \neg\, (x = y) \\
\forall\, x : X; S : \mathbb{P}\, X \bullet x \notin S \Leftrightarrow \neg\, (x \in S) \\
\hline
\end{array}
$$

Description

Inequality is a relation between values of the same type. The predicate $x \neq y$ denotes true when $x = y$ denotes false.

Non-membership is a relation between values of a certain type and sets of values of that type. The predicate $x \notin S$ denotes true when $x \in S$ denotes false.

Name

$\varnothing$ – Empty Set

$\subseteq$ – Subset relation

$\subset$ – Proper subset relation

$\mathbb{P}_1$ – Non-empty subsets

Definition

$$\varnothing[X] == \{ x : X \mid false \}$$

fixity **rel** $\_ \subseteq \_$
fixity **rel** $\_ \subset \_$

$$[X]$$
$$\_ \subseteq \_, \_ \subset \_ : \mathbb{P} X \leftrightarrow \mathbb{P} X$$
$$\forall S, T : \mathbb{P} X \bullet$$
$$(S \subseteq T \Leftrightarrow (\forall x : X \bullet x \in S \Rightarrow x \in T)) \land$$
$$S \subset T \Leftrightarrow S \subseteq T \land S \neq T)$$

$$\mathbb{P}_1 X == \{ S : \mathbb{P} X \mid S \neq \varnothing \}$$

Description

The empty set of values of a certain type is the set of values of that type that has no members.

If S and T are sets of values of the same type, then $S \subseteq T$ is a predicate denoting true if and only if every member of S is a member of T. The empty set of values of a certain type is a subset of every set of values of that type.

If S and T are sets of values of the same type, then $S \subset T$ is a predicate denoting true if and only if every member of S is a member of T and S and T are not equal. If S is a proper subset of T, then it is also a subset of T. The empty set of values of a certain type is a proper subset of every non-empty set of values of that type.

If X is a set, then $\mathbb{P}_1 X$ is the set of all non-empty subsets of X. $\mathbb{P}_1 X$ is a proper subset of $\mathbb{P} X$.

Name

$\cup$ – Set union

$\cap$ – Set intersection

$\setminus$ – Set difference

Definition

fixity **leftfun** 0 $\_ \cup \_$
fixity **leftfun** 0 $\_ \cap \_$
fixity **leftfun** 0 $\_ \setminus \_$

$$[X]$$

$$\_\cup\_, \_\cap\_, \_\setminus\_ : \mathbb{P} X \times \mathbb{P} X \longrightarrow \mathbb{P} X$$

$$\forall S, T : \mathbb{P} X \bullet$$
$$S \cup T = \{ x : X \mid x \in S \lor x \in T \} \land$$
$$S \cap T = \{ x : X \mid x \in S \land x \in T \} \lor$$
$$S \setminus T = \{ x : X \mid x \in S \land x \notin T \}$$

Description

The union of two sets of values of the same type is the set of values that are members of either set.

The intersection of two sets of values of the same type is the set of values that are members of both sets.

The difference of two sets of values of the same types is the set of values that are members of the first set but not members of the second.

Name

$\cup$ – Generalized union

$\cap$ – Generalized intersection

Definition

$$[X]$$

$$\cup, \cap : \mathbb{P}(\mathbb{P} X) \longrightarrow \mathbb{P} X$$

$$\forall A : \mathbb{P}(\mathbb{P} X) \bullet$$
$$\cup A = \{ x : X \mid (\exists S : A \bullet x \in S) \} \land$$
$$\cap A = \{ x : X \mid (\forall S : A \bullet x \in S) \}$$

Description

The generalised union of a set of sets of values of the same type is the set of values of that type that are members of at least one of the sets.

The generalised intersection of a set of sets of values of the same type is the set of values of that type that are members of every one of the sets.

Name

first, *second* – Projection functions for ordered pairs

Definition

$$
\begin{array}{|l}
\hline
[X, Y] \\
\mathit{first} : X \times Y \longrightarrow X \\
\mathit{second} : X \times Y \longrightarrow Y \\
\hline
\forall x : X; y : Y \bullet \\
\quad \mathit{first}(x, y) = x \land \\
\quad \mathit{second}(x, y) = y \\
\hline
\end{array}
$$

Description

For any ordered pair (x, y), $\mathit{first}(x, y)$ is x and $\mathit{second}(x, y)$ is y.

If p is of type $X \times Y$, then $p = (\mathit{first}\ p, \mathit{second}\ p)$.

D.2 Relations

Name

$\leftrightarrow$ – Binary relations

$\mapsto$ – Maplet

Definition

fixity **leftfun** 0 $\_ \leftrightarrow \_$
fixity **leftfun** 0 $\_ \mapsto \_$

$$X \leftrightarrow Y == \mathbb{P}(X \times Y$$

$$[X, Y]$$
$$\_ \mapsto \_ : X \times Y \longrightarrow X \times Y$$
$$\forall x : X; y : Y \bullet$$
$$x \mapsto y = (x, y)$$

Description

$X \leftrightarrow Y$ is the set of all sets of ordered pairs whose first members are members of X and whose second members are members of Y. To declare $R : X \leftrightarrow Y$ is to say that R is such a set of ordered pairs.

The maplet forms an ordered pair from two values, so if x is of type X and y is of type Y, then $x \mapsto y$ is of type $X \times Y$. $x \mapsto y$ is thus just another notation for (x, y).

Name

dom, ran – Domain and range of a relation

Definition

$$[X, Y]$$

$$\mathrm{dom} : (X \leftrightarrow Y) \longrightarrow \mathbb{P}\, X$$
$$\mathrm{ran} : (X \leftrightarrow Y) \longrightarrow \mathbb{P}\, Y$$

$$\forall R : X \leftrightarrow Y \bullet$$
$$\mathrm{dom}\, R = \{ x : X; y : Y \mid (x \mapsto y) \in R \bullet x \} \land$$
$$\mathrm{ran}\, R = \{ x : X; y : Y \mid (x \mapsto y) \in R \bullet y \}$$

Description

The domain of a relation R is the set of first members of the ordered pairs in R. If R is of type $X \leftrightarrow Y$, the domain of R is of type $\mathbb{P}\, X$. If R is an empty relation, then its domain is an empty set.

The range of a relation R is the set of second members of the ordered pairs in R. If R is of type $X \leftrightarrow Y$, the domain of R is of type $\mathbb{P}\, Y$. If R is an empty relation, then its range is an empty set.

Name

id – Identity relation

$\fatsemi$ – Relational composition

$\circ$ – Backward relational composition

Definition

$$\mathrm{id}\, X == \{ x : X \bullet x \mapsto x \}$$

fixity **leftfun** 0 $\_ \fatsemi \_$
fixity **leftfun** 0 $\_ \circ \_$

$$[X, Y, X]$$

$$\_ \fatsemi \_ : (X \leftrightarrow Y) \times (Y \leftrightarrow Z) \rightarrow (X \leftrightarrow Z)$$
$$\_ \circ \_ : (Y \leftrightarrow Z) \times (X \leftrightarrow Y) \rightarrow (X \leftrightarrow Z)$$

$$\forall R : X \leftrightarrow Y; S : Y \leftrightarrow Z \bullet$$
$$R \fatsemi S = S \circ R = \{ x : X; y : Y; z : Z \mid (x \mapsto y) \in R \land (y \mapsto z) \in S \bullet x \mapsto z \}$$

Description

The identity relation on a set X is the relation that relates every member of X to itself. Its type is $X \leftrightarrow X$. The identity relation on an empty set is an empty relation.

The relational composition of a relation $R : X \leftrightarrow Y$ and $S : Y \leftrightarrow Z$ is a relation of type $X \leftrightarrow Z$ formed by taking all the pairs (x, y) of R whose second members are in the domain of S, and relating x to every member of Z that y is related to by S.

The backward composition of S and R is the same as the composition of R and S.

Name

$\lhd$ – Domain restriction

$\rhd$ – Range restriction

Definition

fixity **leftfun** 0 $\_ \lhd \_$
fixity **leftfun** 0 $\_ \rhd \_$

$$
\begin{array}{|l}
[X, Y] \\
\hline
\_ \lhd \_ : \mathbb{P}\, X \times (X \leftrightarrow Y) \rightarrow (X \leftrightarrow Y) \\
\_ \rhd \_ : (X \leftrightarrow Y) \times \mathbb{P}\, Y \rightarrow (X \leftrightarrow Y)' \\
\hline
\forall S : \mathbb{P}\, X; R : X \leftrightarrow Y \bullet \\
\quad S \lhd R = \{ x : X; y : Y \mid x \in S \land (X \mapsto y) \in R \bullet x \mapsto y \} \\
\forall R : X \leftrightarrow Y; T : \mathbb{P}\, Y \bullet \\
\quad R \rhd T = \{ x : X; y : Y \mid (x \mapsto y) \in R \land y \in T \bullet x \mapsto y \} \\
\hline
\end{array}
$$

Description

The domain restriction of a relation $R : X \leftrightarrow Y$ by a set $S : \mathbb{P}\, X$ is the set of pairs in R whose first members are in S. $S \lhd R$ is a subset of R, and its domain is a subset of S.

The range restriction of a relation $R : X \leftrightarrow Y$ by a set $T : \mathbb{P}\, Y$ is the set of pairs in R whose second members are in T. $R \rhd T$ is a subset of R, and its range is a subset of T.

Name

$\lhd$ – Domain anti-restriction

$\rhd$ – Range anti-restriction

Definition

fixity **leftfun** 0 $\_ \lhd \_$
fixity **leftfun** 0 $\_ \rhd \_$

$$
\begin{array}{|l}
[X, Y] \\
\hline
\_ \lhd \_ : \mathbb{P} X \times (X \leftrightarrow Y) \rightarrow (X \leftrightarrow Y) \\
\_ \rhd \_ : (X \leftrightarrow Y) \times \mathbb{P} Y \rightarrow (X \leftrightarrow Y) \\
\hline
\forall S : \mathbb{P} X; R : X \leftrightarrow Y \bullet \\
\quad S \lhd R = \{ x : X; y : Y \mid x \notin S \land (x \mapsto y) \in R \bullet x \mapsto y \} \\
\forall R : X \leftrightarrow Y; T : \mathbb{P} Y \bullet \\
\quad R \rhd T = \{ x : X; y : Y \mid (x \mapsto y) \in R \land y \notin T \bullet x \mapsto y \} \\
\hline
\end{array}
$$

Description

The domain anti-restriction of a relation $R : X \leftrightarrow Y$ by a set $S : \mathbb{P} X$ is the set of pairs in R whose first members are not in S. $S \lhd R$ is a subset of R, and its domain contains no members of S.

The range anti-restriction of a relation $R : X \leftrightarrow Y$ by a set $T : \mathbb{P} Y$ is the set of pairs in R whose second members are not in T. $R \rhd T$ is a subset of R, and its range contains no members of T.

Name

$\_^\sim$ – relational inversion

Definition

fixity **leftfun** 0 $\_^\sim\_$

$$[X, Y]$$
$$\_^\sim : (X \leftrightarrow Y) \rightarrow (Y \leftrightarrow X)$$
$$\forall R : X \leftrightarrow Y \bullet$$
$$R^\sim = \{ x : X; y : Y \mid (x \mapsto y) \in R \bullet y \mapsto x \}$$

Description

The inverse of a relation is the relation obtained by reversing every ordered pair in the relation.

Name

$\_(\!|\_|\!)$ – Relational image

Definition

fixity **leftfun** 0 $\_(\!|\_|\!)$

$$[X, Y]$$

$$\_(\!|\_|\!) : (X \leftrightarrow Y) \times \mathbb{P}\, X \mapsto \mathbb{P}\, Y$$

$$\forall R : X \leftrightarrow Y; S : \mathbb{P}\, X \bullet$$
$$R(\!|S|\!) = \{ x : X; y : Y \mid x \in S \land (x \mapsto y) \in R \bullet y \}$$

Description

The relational image of a set $S : \mathbb{P}\, X$ under a relation $R : X \leftrightarrow Y$ is the set of values of type Y that are related under R to a value in S.

Name

$\_^+$ – Transitive closure

$\_^*$ – Reflexive-transitive closure

Definition

fixity **leftfun** 0 $\_^+\_$
fixity **leftfun** 0 $\_^*\_$

$$[X]$$

$$\_^+, \_^* : (X \leftrightarrow X) \longrightarrow (X \leftrightarrow X)$$

$$\forall R : X \leftrightarrow X \bullet$$
$$R^+ = \cap \{ Q : X \leftrightarrow X \mid R \subseteq Q \land Q \fatsemi Q \subseteq Q \} \land$$
$$R^* = \cap \{ Q : X \leftrightarrow X \mid \mathrm{id}\, X \subseteq Q \land R \subseteq Q \land Q \fatsemi Q \subseteq Q \}$$

Description

The transitive closure of a relation $R : X \leftrightarrow X$ is the relation obtained by relating each member of the domain of R to its images under R, and to anything related to any of its images under R by any number of steps of application of R.

The reflexive transitive closure of a relation $R : X \leftrightarrow X$ is the relation formed by extending the transitive closure of R by the identity relation on X.

D.3 Functions

Name

$\pfun$ – Partial functions

$\fun$ – Total functions

Definition

fixity **leftfun** 0 $\pfun$
fixity **leftfun** 0 $\fun$

$$X \pfun Y == \{ f : X \rel Y \mid (\forall x : X; y_1, y_2 : Y \bullet (x \mapsto y_1) \in f \land (x \mapsto y_2) \in f \Rightarrow y_1 = y_2) \}$$
$$X \fun Y == \{ f : X \pfun Y \mid \mathrm{dom} f = X \}$$

Description

The partial functions from X to Y are a subset of the relations $X \rel Y$. They are distinguished by the property that each x in X is related to at most one y in Y. $X \pfun Y$ is the set of all partial functions from X to Y, and to declare $f : X \pfun Y$ is to say that f is one such partial function.

The total functions from X to Y are a subset of the partial functions $X \pfun Y$. They are distinguished by the property that each x in X is related to exactly one y in Y. $X \fun Y$ is the set of all total functions from X to Y, and to declare $f : X \fun Y$ is to say that f is one such total function. The domain of $f : X \fun Y$ is X.

Name

$\rightarrowtail\!\!\!\!\!\!+$ – Partial injections

$\rightarrowtail$ – Total injections

Definition

fixity **leftfun** 0 $\rightarrowtail\!\!\!\!\!\!+$
fixity **leftfun** 0 $\rightarrowtail$

$$X \rightarrowtail\!\!\!\!\!\!+ Y == \{ f : X \pfun Y \mid (\forall x_1, x_2 : \mathrm{dom}\, f \bullet f(x_1) = f(x_2) \Rightarrow x_1 = x_2) \}$$

$$X \rightarrowtail Y == (X \rightarrowtail\!\!\!\!\!\!+ Y) \cap (X \fun Y)$$

Description

The partial injections from X to Y are a subset of the partial functions $X \pfun Y$. They are distinguished by the property that each y in Y is related to at most one x in X. Thus the inverse of a partial injection is also a partial injection. $X \rightarrowtail\!\!\!\!\!\!+ Y$ is the set of all partial injections from X to Y, and to declare $f : X \rightarrowtail\!\!\!\!\!\!+ Y$ is to say that f is one such partial injection.

The total injections from X to Y are a subset of the partial injections $X \rightarrowtail\!\!\!\!\!\!+ Y$. They are distinguished by the property that each x in X is related to exactly one y in Y. $X \rightarrowtail Y$ is the set of all total injections from X to Y, and to declare $f : X \rightarrowtail Y$ is to say that f is one such total injection.

Name

$\pfun\!\!\!\!\!\rightarrow$ – Partial surjections

$\fun\!\!\!\!\!\rightarrow$ – Total surjections

$\rightarrowtail\!\!\!\!\!\rightarrow$ – Bijections

Definition

fixity **leftfun** 0 $\pfun\!\!\!\!\!\rightarrow$
fixity **leftfun** 0 $\fun\!\!\!\!\!\rightarrow$
fixity **leftfun** 0 $\rightarrowtail\!\!\!\!\!\rightarrow$

$$X \pfun\!\!\!\!\!\rightarrow Y == \{f : X \pfun Y \mid \mathrm{ran} f = Y\}$$
$$X \fun\!\!\!\!\!\rightarrow Y == (X \pfun\!\!\!\!\!\rightarrow Y) \cap (X \fun Y)$$
$$X \rightarrowtail\!\!\!\!\!\rightarrow Y == (X \fun\!\!\!\!\!\rightarrow Y) \cap (X \rightarrowtail Y)$$

Description

The partial surjections from X to Y are a subset of the partial functions $X \pfun Y$. They are distinguished by the property that each y in Y is related to at least one x in X. $X \pfun\!\!\!\!\!\rightarrow Y$ is the set of all partial surjections from X to Y, and to declare $f : X \pfun\!\!\!\!\!\rightarrow Y$ is to say that f is one such partial surjection.

The total surjections from X to Y are a subset of the partial surjections $X \pfun\!\!\!\!\!\rightarrow Y$. They are distinguished by the property that each x in X is related to exactly one y in Y. $X \fun\!\!\!\!\!\rightarrow Y$ is the set of all total surjections from X to Y, and to declare $f : X \fun\!\!\!\!\!\rightarrow Y$ is to say that f is one such total surjection.

The bijections from X to Y are a subset of the total surjections $X \fun\!\!\!\!\!\rightarrow Y$. They are distinguished by the property that each y in Y is related to exactly one x in X. $X \rightarrowtail\!\!\!\!\!\rightarrow Y$ is the set of all bijections from X to Y, and to declare $f : X \rightarrowtail\!\!\!\!\!\rightarrow Y$ is to say that f is one such total bijection.

D.3.1 Name

$\oplus$ – Functional overriding

D.3.2 Definition

fixity **leftfun** 0 $\oplus$

$$[X, Y]$$
$$\_ \oplus \_ : (X \pfun Y) \times (X \pfun Y) \rightarrow (X \pfun Y)$$
$$\forall f, g : X \pfun Y \bullet$$
$$f \oplus g = ((\mathrm{dom}\, g) \domsub f) \cup g$$

Description

If f and g are both functions from X to Y, then the functional overriding of f by g is the function g together with such pairs of f as have first elements different from the first element of any pair in g.

D.4 Numbers and finiteness

Name

$\mathbb{N}$	– Natural numbers
$\mathbb{Z}$	– Integers
$+, -, *, \mathsf{div}, \mathsf{mod}$	– Arithmetic operations
$<, \leq, \geq, >$	– Numerical comparison

Definition

fixity **leftfun** 0 $\_+\_$
fixity **leftfun** 0 $\_-\_$
fixity **leftfun** 1 $\_*\_$
fixity **leftfun** 1 $\_div\_$
fixity **leftfun** 1 $\_mod\_$
fixity **leftfun** 3 $\_-$
fixity **rel** $\_<\_$
fixity **rel** $\_\leq\_$
fixity **rel** $\_\geq\_$
fixity **rel** $\_<>\_$

$[\mathbb{Z}]$

$$
\begin{array}{|l}
\mathbb{N} : \mathbb{P}\,\mathbb{Z} \\
\_+\_, \_-\_, \_*\_ : \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{Z} \\
\_\,\mathsf{div}\,\_, \_\,\mathsf{mod}\,\_ : \mathbb{Z} \times (\mathbb{Z} \setminus \{0\}) \longrightarrow \mathbb{Z} \\
\_- : \mathbb{Z} \longrightarrow \mathbb{Z} \\
\_<\_, \_\leq\_, \_\geq\_, \_>\_ : \mathbb{Z} \leftrightarrow \mathbb{Z} \\
\hline
\mathbb{N} = \{\, n : \mathbb{Z} \mid n \geq 0 \,\} \\
\text{... other definitions omitted...}
\end{array}
$$

Description

The natural numbers are the integers from zero upwards. The type of $\mathbb{N}$ is $\mathbb{P}\,\mathbb{Z}$, since $\mathbb{N}$ is a set of integers. The declaration $n : \mathbb{N}$ makes $\mathbb{Z}$ the type of n, and entails the property $n \geq 0$.

Name

$\mathbb{N}_1$ – Strictly positive integers

succ – Successor function

Definition

$$\mathbb{N}_1 == N \setminus \{0\}$$

$$succ : \mathbb{N} \longrightarrow \mathbb{N}$$
$$\forall n : \mathbb{N} \bullet succ(n) = n + 1$$

Description

The strictly positive numbers $\mathbb{N}$ are the natural numbers except zero.

The successor of any natural number is the next natural number in ascending order.

Name

R^k – Iteration

Definition

$$[X]$$

$$iter : \mathbb{Z} \longrightarrow (X \leftrightarrow X) \longrightarrow (X \leftrightarrow X)$$

$$\begin{aligned}
&\forall R : X \leftrightarrow X \bullet \\
&\quad iter\, 0\, R = \mathrm{id}\, X \land \\
&\quad (\forall k : \mathbb{N} \bullet iter\, (k+1) R = R \fatsemi (iter\, k\, R)) \land \\
&\quad (\forall k : \mathbb{N} \bullet iter\, (-k) R = iter\, k\, (R^{\sim}))
\end{aligned}$$

Description

The iteration of a relation $R : X \leftrightarrow X$ by zero is the identity relation on the set X.

The iteration of a relation $R : X \leftrightarrow X$ by one is the relation R.

The iteration of a relation $R : X \leftrightarrow X$ by an integer greater than one is the composition of R with its iteration by the next lower integer.

The iteration of a relation $R : X \leftrightarrow X$ by an integer less that zero is the iteration of the inverse of R by the corresponding positive integer. Thus the iteration of R by -1 is the inverse of R.

The form: $iter\, k\, R$ is usually written R^k.

Name

.. – Number range

Definition

fixity **leftfun** 0 .. _

$$\_\ldots\_ : \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{P}\,\mathbb{Z}$$

$$\forall\, a, b : \mathbb{Z} \bullet$$
$$a \ldots b = \{\, k : \mathbb{Z} \mid a \leq k \leq b \,\}$$

Description

If a and b are integers, and a is less than b, the number range $a..b$ contains a, b and any integers between.

If a is equal to b, the number range $a..b$ is a singleton set containing a only.

If a is greater than b, the number range $a..b$ is an empty set of integers.

The number range $a..b$ is always finite, and if $b \geq a$ its size is $b - a + 1$.

Name

$\mathbb{F}$ – Finite sets

$\mathbb{F}_1$ – Non-empty finite sets

$\#$ – Number of members of a set

Definition

$$\mathbb{F} X == \{ S : \mathbb{P} X \mid \exists n : \mathbb{N} \bullet \exists f : 1 \,..\, n \rightarrow S \bullet \operatorname{ran} f = S \}$$
$$\mathbb{F}_1 X == \mathbb{F} X \setminus \{\varnothing\}$$

$[X]$

$$\# : \mathbb{F} X \rightarrow \mathbb{N}$$

$$\forall S : \mathbb{F} X \bullet$$
$$\quad \#S = (\mu n : \mathbb{N} \mid (\exists f : 1 \,..\, n \rightarrowtail S \bullet \operatorname{ran} f = S))$$

Description

A set is finite if its members can be put into one-to-one correspondence with the natural numbers from 1 up to some limit. $\mathbb{F} X$ is the set of all finite subsets of X. $\mathbb{F} X$ is a subset of $\mathbb{P} X$. If X is finite, then it is a member of $\mathbb{F} X$.

The non-empty finite subsets of X are the finite subsets of X except the empty set.

The number of members of a finite set is the upper limit of the number range starting with 1 that can be put into one-to-one correspondence with the members of the set.

Name

$\mathrel{+\!\!\!\!\twoheadrightarrow}$ – Finite partial functions

$\rightarrowtail\!\!\!\!\!+\!\!\!\!\twoheadrightarrow$ – Finite partial injections

Definition

fixity **leftfun** 0 $\mathrel{+\!\!\!\!\twoheadrightarrow}$
fixity **leftfun** 0 $\rightarrowtail\!\!\!\!\!+\!\!\!\!\twoheadrightarrow$

$$X \mathrel{+\!\!\!\!\twoheadrightarrow} Y == \{ f : X \mathrel{+\!\!\!\!\rightarrow} Y \mid \operatorname{dom} f \in \mathbb{F}\, X \}$$
$$X \rightarrowtail\!\!\!\!\!+\!\!\!\!\twoheadrightarrow Y == (X \mathrel{+\!\!\!\!\twoheadrightarrow} Y) \cap (X \rightarrowtail\!\!\!\!\!+\!\!\!\!\rightarrow Y)$$

Description

The finite partial functions from X to Y are the partial functions from X to Y whose domains are finite sets.

The finite partial injections from X to Y are the partial injections from X to Y whose domains are finite sets.

Name

min, *max* – Minimum and maximum of a set of numbers

Definition

$$
\begin{array}{l}
min : \mathbb{P}_1 \mathbb{Z} \pfun \mathbb{Z} \\
max : \mathbb{P}_1 \mathbb{Z} \pfun \mathbb{Z} \\
\hline
min = \{ S : \mathbb{P}_1 \mathbb{Z}; m : \mathbb{Z} \mid \\
\qquad m \in S \land (\forall n : S \bullet m \leq n) \bullet S \mapsto m \} \\
max = \{ S : \mathbb{P}_1 \mathbb{Z}; m : \mathbb{Z} \mid \\
\qquad m \in S \land (\forall n : S \bullet m \geq n) \bullet S \mapsto m \}
\end{array}
$$

Description

The minimum of a non-empty set of integers that has a least member is the least member. Sets of integers that have no least member are not in the domain of *min*. If $a \leq b$, $\min a..b = a$.

The maximum of a non-empty set of integers that has a greatest member is the greatest member. Sets of integers that have no greatest member are not in the domain of max. If $a \leq b$, $\max a..b = b$.

D.5 Sequences

Name

seq – Finite sequences

seq_1 – Non-empty finite sequences

iseq – Injective sequences

Definition

$$\text{seq}\, X == \{ f : \mathbb{N} \pfun X \mid \text{dom}\, f = 1 \upto \# f \}$$
$$\text{seq}_1 == \{ f : \text{seq}\, X \mid \# f > 0 \}$$
$$\text{iseq}\, X == \text{seq}\, X \cap (\mathbb{N} \pinj X)$$

Description

A sequence is a finite aggregate of values of the same type in which each value can be identified by its position in the sequence. The formal definition establishes a sequence as a partial function relating the numbers from the set $1..n$ for some n (the domain of the sequence) to the values (the range of the sequence). $\text{seq}\, X$ is the set of all finite sequences of values of type X. The declaration $S : \text{seq}\, X$ says that S is one such finite sequence. Since a sequence is a *function* (i.e. a set of ordered pairs), a sequence might be empty, and the function application notation $S\, i$ can be used to denote the element at position i, provided that i is in the domain of the sequence.

$\text{seq}_1\, X$ is the set of all non-empty finite sequences of values of type X. The declaration $s : \text{seq}_1\, X$ says that s is such a non-empty finite sequence. $\text{seq}_1\, X$ is a subset of $\text{seq}\, X$.

$\text{iseq}\, X$ is the set of all injective finite sequences of values of type X. A sequence is injective if no value appears more than once in the sequence. The declaration $S : \text{iseq}\, X$ says that S is such an injective finite sequence. $\text{iseq}\, X$ is a subset of $\text{seq}\, X$.

Name

$<>$ – Sequence brackets
$\frown$ – Concatenation

Definition

$$[X]$$

$$\_ \frown \_ : \operatorname{seq} X \times \operatorname{seq} X \longrightarrow \operatorname{seq} X$$

$$\forall s, t : \operatorname{seq} X \bullet$$
$$s \frown t = s \cup \{ n : \operatorname{dom} t \bullet n + \#s \mapsto t(n) \}$$

Description

The brackets $< >$ can be used for enumerated sequences. The empty enumeration is the empty function. A singleton enumeration is the function that maps 1 to the element in the enumeration. The function that extends an enumeration is the concatenation function.

Concatenation is a function of a pair of sequences of values of the same type that denotes a sequence that begins with the first sequence and continues with the second. Either or both of the sequences might be empty. If either sequence is empty, the result is the other sequence.

Name

head, *last*, *tail*, *front* – Sequence decomposition

Definition

$$
\begin{array}{|l}
\hline
[X] \\
head, last : \mathrm{seq}_1\, X \longrightarrow X \\
tail, front : \mathrm{seq}_1\, X \longrightarrow \mathrm{seq}\, X \\
\hline
\forall\, s : \mathrm{seq}_1\, X \bullet \\
\quad head\ s = s(1) \land \\
\quad last\ s = s(\#s) \land \\
\quad tail\ s = (\lambda\, n : 1 \mathinner{..} \#s - 1 \bullet s(n+1)) \land \\
\quad front\ s = (1 \mathinner{..} \#s - 1) \lhd s \\
\hline
\end{array}
$$

Description

If S is a non-empty sequence of values of type X, then *head* S is the value of type X that is first in the sequence. Empty sequences are not in the domain of *head*.

If S is a non-empty sequence of values of type X, then *last* S is the value of type X that is last in the sequence. Empty sequences are not in the domain of *last*.

If S is a non-empty sequence of values of type X, then *tail* S is the sequence of values of type X obtained from S by discarding the first member. Empty sequences are not in the domain of *tail*.

If S is a non-empty sequence of values of type X, then *front* S is the sequence of values of type X obtained from S by discarding the last member. Empty sequences are not in the domain of *front*.

Name

rev – reverse

Definition

$$[X]$$

$$rev : \operatorname{seq} X \longrightarrow \operatorname{seq} X$$

$$\forall s : \operatorname{seq} X \bullet$$
$$rev\, s = (\lambda n : \operatorname{dom} s \bullet s(\#s - n + 1))$$

Description

The reverse of a sequence is the sequence obtained by taking its members in the opposite order.

Name

$\upharpoonright$ – Filtering

Definition

$$
\begin{array}{|l}
[X] \\
\hline
\_ \upharpoonright \_ : \operatorname{seq} X \times \mathbb{P} X \longrightarrow \operatorname{seq} X \\
\hline
\forall V : \mathbb{P} X \bullet \\
\quad \langle\rangle \upharpoonright V = \langle\rangle \land \\
\quad (\forall x : X \bullet \\
\qquad (x \in V \Rightarrow \langle x \rangle \upharpoonright V = \langle x \rangle) \land \\
\qquad (x \notin V \Rightarrow \langle x \rangle \upharpoonright V = \langle\rangle)) \land \\
\quad (\forall s, t : \operatorname{seq} X \bullet \\
\qquad ((s \frown t) \upharpoonright V = (s \upharpoonright V) \frown (t \upharpoonright V)) \\
\hline
\end{array}
$$

Description

The filter of a sequence of values of type X by a set of values of type X is the sequence obtained from the original by discarding any members that are not in the set.

Name

$\frown/$ – Distributed concatenation

Definition

$$
\begin{array}{|l}
\hline
[X] \\
\frown/ : \operatorname{seq}(\operatorname{seq} X) \longrightarrow \operatorname{seq} X \\
\hline
\frown/\langle\rangle = \langle\rangle \\
\forall s : \operatorname{seq} X \bullet \frown/\langle s \rangle = s \\
\forall q, r : \operatorname{seq}(\operatorname{seq} X) \bullet \\
\quad \frown/(q \frown r) = (\frown/ q) \frown (\frown/ r) \\
\hline
\end{array}
$$

Description

The distributed concatenation of a sequence of sequences of values of type X is a sequence of values of type X obtained by concatenating the lesser sequences in the order in which they appear in the greater sequence.

Name

disjoint – Disjointness

partition – Partitions

Definition

$$[I, X]$$

$$\mathsf{disjoint}\_ : \mathbb{P}(I \pfun \mathbb{P} X)$$

$$\_ \mathsf{partition} \_ : (I \pfun \mathbb{P} X) \rel \mathbb{P} X$$

$$\forall S : I \pfun \mathbb{P} X; T : \mathbb{P} X \bullet$$
$$(\mathsf{disjoint}\ S \Leftrightarrow$$
$$(\forall i, j : \mathrm{dom}\ S \mid i \neq j \bullet S(i) \cap S(j) = \varnothing)) \land$$
$$(S\ \mathsf{partition}\ T \Leftrightarrow$$
$$\mathsf{disjoint}\ S \land \cup \{ i : \mathrm{dom}\ S \bullet S(i) \} = T)$$

Description

An indexed family of sets is disjoint if no two members having distinct indexes have any members in common.

An indexed family S of sets partition a set T if S is disjoint and the union of all the members of S is T.

D.6 Bags

Name

bag – Bags

count – Multiplicity

$\in$ – Bag membership

Definition

$$\mathrm{bag}\, X == X \pfun \mathbb{N}_1$$

$$[X]$$

$$\begin{array}{l} count : \mathrm{bag}\, X \rightarrowtail\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\rightarrow (X \fun \mathbb{N}) \\ \_ \in \_ : X \rightleftarrows \mathrm{bag}\, X \\ \hline \forall x : X; B : \mathrm{bag}\, X \bullet \\ \quad count\, B = (\lambda x : X \bullet 0) \oplus B \land \\ \quad x \in B \Leftrightarrow x \in \mathrm{dom}\, B \end{array}$$

Description

A bag represents an aggregate in which order is not important, but in which a given value can occur several times. A bag of values of type X is a function whose domain is a subset of X and whose range is a set of strictly positive natural numbers.

The count of a bag of values of type X is a function that extends the bag function by relating every member of X that is not is the domain of the bag to zero.

A value $x : X$ is said to be in $B : bagX$ if and only if x is in the domain of B.

Name

$\uplus$ – Bag union

Definition

$$[X]$$

$$\_ \uplus \_ : \operatorname{bag} X \times \operatorname{bag} X \longrightarrow \operatorname{bag} X$$

$$\forall B, C : \operatorname{bag} X; x : X \bullet$$
$$count\ (B \uplus C) x = count\ B\ x + count\ C\ x$$

Description

The bag union of two bags is the bag that relates every member of the domain of either bag to the sum of its occurrences in the two bags.

Name

items – Bag of elements of a sequence

Definition

$$[X]$$

$$items : \operatorname{seq} X \longrightarrow \operatorname{bag} X$$

$$\forall s : \operatorname{seq} X; x : X \bullet count\ (items\ s)x = \#\{ i : \operatorname{dom} s \mid s(i) = x \}$$

Description

The items of a sequence of values of type X is a bag such that the range of the sequence and the domain of the bag are the same, and the each value in the domain of the bag is related to the number of indexes in the sequence at which that value occurred.

□

E Interchange format – Normative Annex

Notes on this section of the Z Standard

Section title: Interchange format
Section editor: Jonathan Hammond (Trevor King)
Contributions by: Trevor King, ... *(others to be added)*
Notes: Conforms to new syntax, lexis and toolkit.
Source file: icformat.tex
Most recent update: 30th June 1995
Formatted: 3rd July 1995

E.1 Introduction

The **Interchange Format** defines a portable representation of Z, allowing Z documents to be transmitted between different products or machines. The most suitable means of communication is the use of text files in which the character set is limited for portability reasons. The Interchange Format defines a syntax for such text files.

The basis for the Interchange Format is the ISO Standard Generalized Markup Language (SGML). SGML permits the structure of texts to be represented and encoded in a standard form, convenient for storage, editing, retrieval and processing. The SGML Standard is defined in [12]. A general description of the aims and principles of SGML, together with an annotated version of the standard, is included in *The SGML Handbook* by C. F. Goldfarb [9]. Case studies and applications in SGML are described in the work of the Text Encoding Initiative reported in [22].

The structure of this Appendix is as follows:

- section E.2 describes the scope of the Interchange Format — i.e. the facilities offered by the Format;
- section E.3 contains an informal description of SGML;
- section E.4 defines the Interchange Format;
- section E.5 presents explanatory material and examples of the use of the Interchange Format.

E.2 Scope of the Interchange Format

The Interchange Format allows a distinction to be made between formal text and other text included in a Z document. The Interchange Format does not prescribe the structure of all parts of a Z document; in particular it does not define the internal structure of *informal* text.

As one possible application of the Interchange Format is to transmit a Z document for Z syntax checking, the format is sufficiently liberal to permit syntactically-incorrect Z to be written. The format thus prescribes markup only for the higher levels of the Z syntax hierarchy. In most cases this is at the

level of a Z paragraph, although for axiomatic and 'boxed' definitions there is scope for creating a more detailed markup if desired (e.g. in order to indicate a presentation format).

For a Z document to be syntactically correct when written in the Interchange Format, it must conform at the higher levels to the markup defined in this Appendix, and at the lower levels (e.g. predicate or expression level) to the Z Concrete Syntax, with all mathematical symbols replaced by the alphanumeric representations discussed in Section E.4.3.

The Interchange Format also provides markup for requirements which are additional to the prime requirement for encoding the *structure* of the Z in a document. The following additional requirements are accommodated:

- identification of informal Z fragments, i.e. Z fragments which do not belong to the *formal* part of a Z document;
- indication of particular presentation formats, e.g. whether a vertical or horizontal style should be employed for a schema definition;
- allocation of identifiers to Z paragraphs, e.g. so that associations between Z operation schemas and data-flow diagrams can be made, or so that Z definitions can be indexed;
- logical grouping of Z paragraphs independently of the positions they occupy in the document, e.g. so that the group can be considered as a unit for type-checking purposes, or that 'units of conservative extension' can be identified for subsequent processing by a proof tool;
- labelling of 'stacked' predicates in an axiomatic or 'boxed' definition.

E.3 Introduction to SGML

This section provides an introduction to SGML, sufficient for the understanding of the definition of the Interchange Format in Section E.4. More comprehensive descriptions of SGML are given in [12] and [9].

Examples of text written in SGML are printed with a fixed-width font (the `tt` font in LaTeX) as follows:

```
<tag> text </tag>
```

E.3.1 SGML Element Definitions

Structures are described in the Interchange Format by means of SGML **elements**. Elements are delimited by start-tags and end-tags. A start-tag is of the form <name>, where name is the generic identifier of the delimited element. The end-tag is of the form </name>. For example, a particular Z given set definition may be written in the Interchange Format as:

```
<givendef> NAME, DATE </givendef>
```

The internal structure of a general SGML element is itself defined in SGML by means of a formal SGML **element declaration**. The components of an element declaration are:

1. the name of the element;
2. two characters (separated by a space) which specify the **minimisation rules** for the element;
3. the **content model** of the element.

The minimisation rules indicate whether the start-tags or end-tags may be omitted in instances of the element. The first character in the pair corresponds to the start-tag and the second to the end-tag. The character '-' or 'o' indicates that the corresponding tag respectively must be present or may be omitted.

The content model specifies what any occurrences of the element may legitimately contain. Contents may be specified in terms of other elements and special reserved words. Ultimately all elements consist of 'parsed character data' (represented in element declarations by the reserved word `#PCDATA`), which contains any valid character data but *not* further elements. Further structural information concerning elements which are constituents of the declared element is provided by the use of **occurrence indicators** and **group connectors**.

Occurrence indicators define how many times a constituent element may occur in instances of the defined element and are placed at the end of the constituent element. The following occurrence indicators are used in this Appendix:

- a question mark (?) indicates that the preceding element occurs at most once;
- an asterisk (*) indicates that the preceding element may be absent or occurs one or more times;
- a plus sign (+) indicates that the preceding element occurs one or more times.

Group connectors specify the ordering of constituent elements. The following connectors are used in this Appendix:

- a vertical bar (|) indicates that only one of the components it connects may appear;
- a comma (,) indicates that the components must appear in that order.

For example the element definition for a Z schema declaration is given as:

```
<!ELEMENT schemadef    - -
          ((#PCDATA | sub | mixedname)+, formals?, decpart?, axpart?) >
```

Occurrences of this element thus consist of a sequence of parsed character data, subscript and mixed name elements (representing the name of the schema), followed by an optional element which holds the formal parameters of the definition, followed by elements representing the optional declaration and axiomatic parts of the schema definition. The start-tag and end-tag of the schema definition must both be present.

E.3.2 SGML attribute declarations

In SGML, **attributes** are used to provide information associated with elements. The Interchange Format employs attributes to encode layout information and other information which is not considered to be part of the *structure* of a Z specification. For example, the Interchange Format defines a 'style' attribute for schema definitions which permits an indication of whether the definition should be in vertical or horizontal form. An occurrence of a 'schemadef' element may thus contain an **attribute-value pair** inside the element's start-tag; for example:

```
<schemadef style=vert> S ...  </schemadef>
```

An SGML **attribute declaration** specifies the name(s) of the element(s) to which the attributes are attached, followed by a list of rows, each of which consists of the name of the attribute being declared, its type, and an optional default value. A type may be given as a collection of explicit values, or as one of the following special keywords:

`CDATA`	the attribute value may contain any valid character data and must be delimited by double quotation marks;
`ID`	indicates that a unique identifying value will be supplied for each instance of the element;
`NMTOKEN`	the attribute value is a name token (i.e. any alphanumeric string);
`NUMBER`	the attribute value is a number.

The default value for an attribute may be denoted as one of the set of explicit values defined for an attribute; alternatively it may be one of the following special values:

`#IMPLIED`	a value need not be supplied;
`#REQUIRED`	a value must be supplied.

E.3.3 SGML entities

An **SGML entity** is a named part of a marked-up document. An example of an entity declaration is:

```
<!ENTITY  ZBS      'Z Standard, version 1.0' >
```

References to entities are contructed by prefixing the name of the entity with an ampersand character (&) and delimiting the end of the name with a semicolon, space or end-of-file. Here is an example of an entity reference:

```
We are now in a position to issue the &ZBS;.
```

The entity reference in this document fragment would be expanded by an SGML parser as:

> We are now in a position to issue the Z Standard, version 1.0.

In the Interchange Format, SGML entities are used to represent certain Z symbols or words. The associations between the alphanumeric representation of mathematical symbols or words and their local codes should be defined in SGML entity declarations. However – since local word processor codes may differ – section E.4.3 presents a scheme in which the entity names used in the Interchange Format are listed against the usual presentation format of the corresponding Z symbols or words.

E.4 Definition of the Interchange Format

This section presents the definition of the Interchange Format as a collection of SGML declarations. Explanatory material and examples of the use of the Interchange Format are also given in Section E.5.

An SGML Document Type Definition (DTD) defines the syntax of SGML-conformant documents in a style which is readable by SGML parsers. The Interchange Format does not warrant a full DTD for two reasons:

- the format does not specify the structure of the *informal* text in a Z document;
- the entity declarations are implementation-dependent.

A DTD consists of a header, followed by a body containing the element declarations, attribute declarations and entity declarations. The definition of the Interchange Format presented in this Section may be considered as the partial body of a DTD (*partial* because the entity declarations are not given explicitly); it is also equivalent to a definition in BNF of the structure of the Interchange Format. Newlines in a Z document are not significant in the translation to the Interchange Format except where they serve to separate predicates or declarations.

It is unlikely that this Interchange Format could ever accommodate every function required by its users. However, any collection of SGML declarations (such as those which define this Interchange Format) may be replaced or enhanced by the pre-insertion of additional SGML declarations. Such a 'customisation' of the Interchange Format would be acceptable by SGML parsers.

E.4.1 Element declarations

These declarations define the higher-level structure of the Z paragraphs in a Z document written in the Interchange Format. They correspond closely to the appropriate constructs of the Z Concrete Syntax. Note that no element minimisation options are offered; the start and end tags must both be present for each element instance.

There are two top-level elements:

- the `Z` element represents a sequence of Z paragraphs;
- the `informalZ` element represents a fragment of mathematics which does not belong to the *formal* part of the specification document.

The `Z` element contains a (possibly empty) sequence of individual Z paragraph elements which constitute (part of) a formal specification.

```
<!ELEMENT Z     - -
      (fixity | givendef | axdef | constraint
          | schemadef | gendef | abbrevdef
          | goal | structsetdef)* >
```

The **informalZ** element defines a fragment of mathematics which is not to be considered as a *formal* part of the specification; these fragments may be **Z** elements, **declaration** elements, or a sequence of parsed character data, subscript elements and mixed name elements (corresponding to unstructured fragments of mathematics).

```
<!ELEMENT informalZ    - -
      (Z | declaration | (#PCDATA | sub | mixedname)+) >
```

Elements representing goals, top-level constraints, declarations, abbreviation definition bodies, predicates, fixity template expressions, given set definitions, declarations of formal parameters, subscripts and operator names in fixity template definitions all consist of an unstructured sequence of parsed character data, subscript elements and mixed name elements.

```
<!ELEMENT (goal | constraint | declaration
| body | predicate |exp | givendef
| formals | sub | namearg)    - -
                                (#PCDATA | sub | mixedname)+ >
```

Elements representing axiomatic definitions, schema definitions and generic definitions contain **decpart** and **axpart** elements; the latter element is optional, as is also the **decpart** element for schema definitions. Schema definitions and generic definitions may contain a **formals** element (representing the declaration of formal parameters). The name introduced by a schema definition is modelled as an unstructured sequence of parsed character data, subscript elements and mixed name elements.

```
<!ELEMENT axdef    - -    (decpart, axpart?) >

<!ELEMENT schemadef    - -
          ((#PCDATA | sub | mixedname)+, formals?, decpart?, axpart?) >

<!ELEMENT gendef    - -    (formals?, decpart, axpart?) >
```

The element representing an abbreviation definition consists of the name introduced by the definition (which is modelled as an unstructured sequence of parsed character data, subscript elements and mixed name elements), an optional **formals** element which represents the declaration of formal parameters, and a **body** element which represents the right-hand side of the abbreviation definition.

```
<!ELEMENT abbrevdef    - -
    ((#PCDATA | sub | mixedname)+, formals?, body) >
```

The element representing a structured set definition consists of the name introduced by the definition (which is modelled as an unstructured sequence of parsed character data, subscript elements and mixed name elements) and a non-empty sequence of **branch** elements.

```
<!ELEMENT structsetdef    - -
     ((#PCDATA | sub | mixedname)+, branch+) >
```

The element representing a branch of a structured set definition consists of a constructor name (which is modelled as an unstructured sequence of parsed character data, subscript elements and mixed name elements) and an `exp` element.

```
<!ELEMENT branchf     - -    ((#PCDATA | sub | mixedname)+, exp) >
```

The element representing the axiomatic part of a 'boxed' definition consists of a sequence of `predicate` elements, representing the predicates which are intended to be separated by the weakly-binding conjunction denoted by significant newlines.

```
<!ELEMENT axpart      - -     (predicate+) >
```

The element representing the declaration part of a 'boxed' definition consists of a sequence of `declaration` elements, each representing a collection of declarations which is separated from other such collections by significant newlines.

```
<!ELEMENT decpart     - -     (declaration+) >
```

The element which represents a fixity statement consists of an unstructured sequence of parsed character data, subscript elements and mixed name elements (modelling the first part of the operator name introduced by the statement) and a (possibly empty) sequence of `namearg` and `exparg` elements which represents the rest of the fixity template. Each `exparg` element consists of three `exp` elements followed by an unstructured sequence of parsed character data, subscript elements and mixed name elements which models part of the operator name.

```
<!ELEMENT fixity      - -
            ((#PCDATA | sub | mixedname)+, (namearg | exparg)*) >

<!ELEMENT exparg      - -
    (exp, exp, exp, (#PCDATA | sub | mixedname)+) >
```

The element which represents a mixed name consists of parsed character data. A mixed name element must be employed in cases where a Z name consists of more than one entity reference or of a mixture of entity references and normal characters.

```
<!ELEMENT mixedname     - -           #PCDATA >
```

E.4.2 Attribute declarations

The attribute declarations permit the association of additional information with occurrences of elements in a Z document written in the Interchange Format.

The attributes `id` and `group` permit identification and logical grouping of Z paragraphs respectively.

```
<!ATTLIST
    (givendef | axdef | constraint | schemadef | gendef
     | goal | abbrevdef | structsetdef)
     id    ID    #IMPLIED
     group    NMTOKEN    #IMPLIED >
```

The attributes `style` and `purpose` define the layout and intended use of a schema definition respectively.

```
<!ATTLIST  schemadef
      style    (vert | horiz)    vert
      purpose    (state | operation | datatype)    #IMPLIED >
```

The attribute `label` permits informal annotation of each member of the 'stack' of predicates which constitutes the axiomatic part of a boxed definition.

```
<!ATTLIST  predicate      label    CDATA    #IMPLIED >
```

The following attributes are used with the `fixity` element:

- the attribute `category` indicates whether the defined operator is a relation, left-associative function or right-associative function;
- the attribute `prec` indicates the binding priority of the defined function (this attribute is not required if the value of `category` is `rel`);
- the attributes `firstarg` and `lastarg` indicate the kind of first and last argument (if any) respectively in the fixity statement.

```
<!ATTLIST  fixity
      category    (leftfun | rightfun | rel)    #REQUIRED
      prec    #NUMBER    #IMPLIED
      (firstarg | lastarg)    (normal | type)    #IMPLIED >
```

The attribute `midarg` of the `namearg` element indicates the kind of argument which appears just before the part of the operator name presented in the `namearg` element.

```
<!ATTLIST  namearg
      midarg    (normal | type)    #REQUIRED >
```

E.4.3 Entity declarations

Editor's note: This subsection lists examples of entity definitions that might be required for the toolkit. It will be updated when a final version of the toolkit has been developed.

Entity declarations are used to define alphanumeric denotations for certain Z symbols and words. The entity declarations for the Interchange Format are not presented in the conventional SGML format because of the dependence of the internal representation of mathematical symbols or words on the implementation of each user's Z document processor. The mode of declaration used in this Standard is to present tables which record the association of each entity name with the corresponding mathematical symbol or word. These tables are presented in the Lexis Section of this Standard.

Many of the entity names used in the tables have already been defined as standard in Appendix D of [12].

In order to encode names which consist of more than one entity reference, or of a mixture of entity references and normal text, "mixed name" elements must be employed. For example, the application of the function Δ to the variable *State* (ie Δ *State*) is encoded as `&Delta; State` (or `&Delta;State` or `&Delta State`), but the schema name Δ*State* must be encoded as any of these alternatives enclosed within `mixedname` element tags. The use of the `mixedname` element indicates that the contents are to be considered as a single name.

Z users may create additional entity declarations to cater for new symbols introduced in their specification documents. For example, assume that the new symbol ⊘ is to be used in a Z document. The author of the document must create an entity declaration of the form

```
<!ENTITY   oslash    SDATA  " oslash"
  --o enclosing a solidus-->
```

This declaration gives the name `oslash` to the entity which represents the symbol and identifies the local code which generates the presentation format ⊘ of the symbol. As it is unlikely that *other* systems possess this code, a description of the presentation format of the symbol is given as a comment in the entity declaration. Any receivers of the Z document may then establish a suitable local code for this symbol in their respective systems.

Entity definitions for the toolkit

Entity definitions are provided only for the following classes of toolkit member:

- non-alphanumeric members (apart from the addition (+) and multiplication (*) symbols, as it is assumed that these symbols are reasonably portable);
- relations with alphanumeric names (as these are customarily presented in sans-serif font);
- 'type constructors' (ie generic constants with set values) with alphanumeric names (as these are customarily presented in Roman font);
- the functions dom and ran (as these are customarily presented in Roman font).

E.5 Examples

This section presents examples of the use of the Interchange Format. Thses examples are carefully chosen to cover the more obscure aspects of the Format. The areas covered are indicated in the subsection headings.

E.5.1 Declaring infix identifiers

Consider the following axiomatic definition, which declares a relation *isTwice* which is intended to be used in an infix manner:

SYNTAX REL NORMAL *isTwice* NORMAL

$$\begin{array}{|l} \_isTwice\_ : \mathbb{N} \leftrightarrow \mathbb{N} \\ \hline \forall i, j : \mathbb{N} \bullet i \; isTwice \; j \Leftrightarrow i = 2 * j \end{array}$$

This can be encoded in the Interchange Format as

```
<Z>
<fixity  category=rel  firstarg=normal  lastarg=normal>
isTwice </fixity>

<axdef>
<decpart>
<declaration>
  _isTwice_: &Nat; &rel; &Nat;
</declaration> </decpart>
<axpart>
  <predicate>
  &forall; i, j: &Nat; &bull; i isTwice j &iff; i = 2*j
</predicate> </axpart>
</axdef>
</Z>
```

E.5.2 Subscripts

The axiomatic definition

$$\begin{array}{|l} a_1, \, a_3 : \mathbb{N} \\ \hline a_3 \; isTwice \; a_1 \end{array}$$

is encoded in the Interchange Format as:

```
<Z> <axdef>
<decpart>
  <declaration>
    a<sub> 1 </sub>, a<sub> 3 </sub>: &Nat;
</declaration> </decpart>
<axpart>
  <predicate>
    a<sub> 3 </sub> isTwice a<sub> 1 </sub>
</predicate> </axpart>
</axdef> </Z>
```

E.5.3 Schema definitions and predicate labelling

Consider the following definitions:

$[PERSON, HOUSE]$

Street

$$inhabits : PERSON \pfun HOUSE$$
$$houses : \mathbb{P}\, HOUSE$$

$$houses = \operatorname{ran} inhabits$$
$$\forall h : houses \bullet \#inhabits^{\sim}(\!|\{h\}|\!) \leq 4$$
$$/* \textit{ No house may be occupied by more than 4 persons } */$$

The author of this specification intends to accomplish the following objectives:

- to attach a label to the second predicate in the schema definition;
- to indicate that the schema definition should be displayed in vertical form;
- to indicate (to a specification checker, for example) that the schema *Street* defines the *state* of a system.

These objectives can be attained in the Interchange Format with the following encoding:

```
<Z>
<givendef> PERSON, HOUSE </givendef>

<schemadef style=vert purpose=state> Street
<decpart>
<declaration>
   inhabits: PERSON &fpfun; HOUSE </declaration>
<declaration>
   houses: &pset; HOUSE </declaration>
```

```
<axpart>
  <predicate>
    houses = &ran; inhabits  </predicate>
  <predicate
     label='No house may be occupied by more than 4 persons'>
    &forall; h: houses &bull;
       &num; inhabits &tilde; &limg; &lcub; h &rcub; &rimg;  &le; 4
</predicate> </axpart>
</schemadef>
</Z>
```

E.5.4 Symbols in top-level definitions

Note that in the Interchange Format there are no *entity* representations of the symbols immediately associated with top-level definitions such as structural set definitions and abbreviation definitions. These symbols are subsumed by the element tags for those definitions. For example, consider the following abbreviation definition:

$$n == 5 + x$$

This definition is encoded in the Interchange Format as:

```
<Z> <abbrevdef> n
<body> 5 + x </body> </abbrevdef> </Z>
```

i.e. there is no need in the Interchange Format for an SGML entity which represents the == symbol (although this symbol would of course be employed in the *presentation format* for instances of the `abbreviation` element).

□

F The logical theory of Z – Normative Annex

Notes on this section of the Z Standard

Section title: The Logical Theory of Z
Section editor: Andrew Martin
Original text by: Stephen Brien
Contributions by: Peter Lupton ... *(to be added)*
Source file: logical.tex
Most recent update: 29th June 1995
Formatted: 3rd July 1995

Editor's note: Draft—comments and questions in boxes!

F.1 Preamble

Editor's note: *To be supplied by Peter Lupton.* This chapter defines what the theorems of Z are, discusses the relationship between logic, deductive system, semantics; and soundness. It also describes how other deductive systems (and logics?) are to be derived; what it means for them to be conformant/sound and complete.

Editor's note: This chapter presents a deductive system for Z, a deductive type system for Z and equations for free variables and substitution of terms in Z.

old text:
The deductive system is a Gentzen-style sequent calculus in which sequents are composed of paragraphs and predicates. The rules of the logic are presented in a simplified form. The meta-theorems of the logic (theorems about the rules) permit the extension of the rules into a more practical form.

The loose definition of function application and definite description in the semantics permits a number of interpretations of their meanings. This deductive system is sound with respect to a model in which all well-typed expressions have a value.

F.2 Meta-language

The deductive system will be expressed using the abstract syntax. Some simplifications will be used. In particular, the paragraph keywords **given, gendef, abbr** etc. will be omitted. The concrete syntax

apparently now has a $\mathbf{let}\, x := u \;\mathbf{in}\; t$ construction. The deductive system is better expressed with $(\!|\, x := u \,|\!) \odot t$.

> **Editor's note:** Stephen has $(\!|\, b \,|\!)\ P$ for substitution in predicates, and $b \odot e$ for substitution in expressions. Our abstract syntax doesn't permit the first, BUT some of the semantic equivalences need predicate and expression substitution to be distinguished. I've used $b \odot P$ for the former, for the time being.

F.2.1 Meta-Variables

The meta-variables used in what follows are members of the following syntactic categories:

$$\begin{aligned} \Pi &:: \text{PAR} \\ \Gamma &:: \text{PAR} \ddagger \ldots \ddagger \text{PAR} \\ P, Q, R &:: \text{PRED} \\ x &:: \text{NAME} \\ b, e, f, s, v &:: \text{EXP} \\ S, T &:: \text{SCHEMA} \\ q &:: \text{DECOR} \end{aligned}$$

F.2.2 Sequent

> **Editor's note:** We assume that the Abstract Syntax has been updated so that CONJECTURE = **conj** PAR $\ddagger \cdots \ddagger$ PAR $\vdash$ PRED.

> **Editor's note:** The abstract syntax for PAR is unclear. It doesn't seem to cater for all possibilities. For the time being, this chapter uses Stephen's paragraphs.

The abstract syntax has a paragraph form CONJECTURE, which is a sequent:

$$\Pi_1 \ddagger \cdots \ddagger \Pi_n \vdash P \quad .$$

The meaning of this paragraph (in terms of the semantics) is described elsewhere. For the sequent to be well-formed, the free variables of P must be declared in $\Pi_1 \ddagger \cdots \ddagger \Pi_n$. The preamble above describes what it means for a sequent to be a theorem of Z; the deductive system defines which sequents may be deduced from which other sequents.

This chapter also presents *type judgements* of the form

$$\Gamma \vdash e \mathbin{:} \tau \quad .$$

This sequent means that, in the specification Γ, the expression e is well-typed with type τ. The following proof rules assume that τ is arbitrary, it should be established by pattern matching. For predicates (which do not have a type) we write $\vdash P\ \surd$ to mean that the predicate P is well typed. For paragraphs we write $\vdash P ::$ to indicate that they are well-typed. Though the turnstile is the same

as for the deduction rules, it is used to represent different kinds of relations. These assertions can be distinguished by the syntax of the consequent (the antecedent in both cases being a specification).

Editor's note: *What about provisos-as-judgements? (See Section F.7.)*

F.2.3 Rules

The deductive systems consist of a number of rules for manipulating sequents. Inference rules will be written

$$Rule \mathrel{\widehat=} \frac{Premises}{Conclusion}\ Name\ (Proviso) \quad .$$

The premises are a (possibly empty) list of sequents:

$$Premises \mathrel{\widehat=} Sequent \ldots Sequent \quad .$$

The conclusion is always a single sequent:

$$Conclusion \mathrel{\widehat=} Sequent \quad .$$

The Proviso is a decidable condition on the free variables of the expressions and predicates in the rule. The annotation $\uparrow\downarrow$ indicates that the rule can be applied in both directions—that is, the rule

$$\frac{\Gamma \vdash \Psi}{\Gamma' \vdash \Phi}\ \uparrow\downarrow$$

denotes both of the following inference rules

$$\frac{\Gamma \vdash \Psi}{\Gamma' \vdash \Phi} \quad \text{and} \quad \frac{\Gamma' \Phi}{\Gamma \vdash \Psi} \quad .$$

A rule is *sound* if whenever it is applied to valid premisses, a valid conclusion results. This is defined in the semantics by saying that the set of environments supporting the premisses is a subset of those supporting the conclusion. The rule

$$\frac{S_1 \quad \ldots \quad S_m}{Seq}\ N(P)$$

is sound if and only if

$$P \quad \Rightarrow \quad [\![S_1]\!]^{\mathcal{M}} \cap \ldots \cap [\![S_m]\!]^{\mathcal{M}} \subseteq [\![Seq]\!]^{\mathcal{M}}$$

The following meta-theorem holds for rules in the deductive system:

Theorem F.1 (Sequent-lifting)

The rule $\overline{\Gamma \vdash P}$ *is sound if and only if the sequent* $\Gamma \vdash P$ *is a theorem.*

This meta-theorem states that a theorem can be deduced from no premisses.

The semantic equivalences for substitution are given in tables in later sections. These tables state the semantic equality of various expressions. A theorem which permits the use of semantic-equivalences in proofs is the following.

Theorem F.2 (Semantic-equivalence-lifting) *Given the semantic-equivalence for any predicates or expressions*

$$A \equiv B$$

the following inference rule is sound:

$$\frac{\Delta(A)}{\Delta(B)}$$

F.2.4 wf

The proviso $\mathbf{wf}(par)$ is an abbreviation stating that (par) is well-formed in that

$$\phi(par) \cap \alpha(par) = \varnothing \quad .$$

F.2.5 Proofs

Proofs in the deductive system proceed in the way that is usual for sequent calculi: proofs are developed *backwards,* starting from the sequent which is to be proved. A rule is applied, resulting in fresh sequents which must be proved. This process continues until there are no more sequents requiring proof, in which case the original sequent is now proved.

A completed proof may thus be represented as a tree, with the proved sequent as the root node, and every leaf node containing an empty list of sequents. However, if some of these lists in the leaves are non-empty, then the derivation tree is still useful, although it does not represent a proof, it represents a partial proof.

Theorem F.3 (Tree-squashing) *Suppose that we have the derivation tree:*

$$\frac{S_1 \quad \ldots \quad \dfrac{S_{i1} \quad \ldots \quad S_{im}}{S_i}\,[R_i](P_i) \quad \ldots \quad S_n}{Seq}\,[R](P)$$

where each of the rules R and R_i are sound rules, then the derived rule

$$\frac{S_1 \quad \ldots \quad S_{i1} \quad \ldots \quad S_{im} \quad \ldots \quad S_n}{Seq}\,[R'](P, P_i)$$

is also sound.

F.3 Inference rules

F.3.1 Structural rules

Assumption rules

$$\frac{}{P \vdash P}\ AssumPred$$

$$\frac{}{x := e \vdash x = e}\ AssumDefin(\mathbf{wf}(x := e))$$

$$\frac{}{x : s \vdash x \in s}\ AssumDecl(\mathbf{wf}(x : s))$$

$$\frac{}{S \vdash S}\ SchemaAss(\alpha S \cap \phi S = \varnothing)$$

Paragraph and thinning rules

$$\frac{P \wedge R \vdash Q}{P \ddagger R \vdash Q}\ \uparrow\downarrow PredConj$$

$$\frac{\Gamma \vdash P}{\Pi \ddagger \Gamma \vdash P}\ Thinl$$

$$\frac{\Gamma \vdash P}{\Gamma \ddagger \Pi \vdash P}\ Thinr(\alpha\Pi \cap \phi P = \varnothing)$$

$$\frac{\Gamma_1 \ddagger \Gamma_2 \ddagger \Pi \vdash P}{\Gamma_1 \ddagger \Pi \ddagger \Gamma_2 \vdash P}\ Shift\left(\begin{array}{c}\alpha\Gamma_2 \cap \phi\Pi = \varnothing \\ \alpha\Pi \cap \phi\Gamma_2 = \varnothing\end{array}\right)$$

$$\frac{\Gamma \vdash P}{\Gamma \ddagger [x]T \vdash P}\ D1(\alpha T \cap \phi P = \varnothing)$$

F.3.2 Equality and substitution

$$\frac{}{\Gamma \vdash e = e}\ Refl$$

$$\frac{\Gamma \vdash u = e}{\Gamma \vdash e = u}\ Symm$$

$$\frac{\Gamma \ddagger u = e \vdash v = e}{\Gamma \ddagger u = e \vdash v = u}\ Trans$$

$$\frac{\Gamma \ddagger (\!|\ b\ |\!) \vdash P}{\Gamma \vdash (\!|\ b\ |\!) \odot P}\ \uparrow\downarrow UseBind$$

$$\frac{\Gamma \ddagger (\!|\ b\ |\!) \vdash u = e}{\Gamma \vdash u = b \odot e}\ \uparrow\downarrow EquBind(\alpha b \cap \phi u = \varnothing)$$

F.3.3 Propositional calculus

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \land Q}\ AndI$$

$$\frac{\Gamma \vdash P \land Q}{\Gamma \vdash P}\ AndEr$$

$$\frac{\Gamma \vdash P \land Q}{\Gamma \vdash Q}\ AndEl$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \lor Q}\ OrIr$$

$$\frac{\Gamma \vdash Q}{\Gamma \vdash P \lor Q}\ OrIl$$

$$\frac{\Gamma \vdash P \lor Q \qquad \Gamma \ddagger P \vdash R \qquad \Gamma \ddagger Q \vdash R}{\Gamma \vdash R}\ OrE$$

$$\frac{\Gamma \ddagger P \vdash Q}{\Gamma \vdash P \Rightarrow Q}\ impI$$

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash P \Rightarrow Q}{\Gamma \vdash Q}\ impE$$

$$\frac{\Gamma \vdash \mathsf{false}}{\Gamma \vdash P}\ falseE$$

$$\frac{\Gamma \ddagger \neg P \vdash \mathsf{false}}{\Gamma \vdash P}\ notE$$

$$P \Leftrightarrow Q \equiv P \Rightarrow Q \land Q \Rightarrow P$$

$$\neg P \equiv P \Rightarrow \mathsf{false}$$

$$\mathsf{false} \Rightarrow P$$

$$P \Rightarrow \mathsf{true}$$

Editor's note: Hence, derived laws:

$$\frac{\Gamma \vdash P \Rightarrow \mathsf{false}}{\Gamma \vdash \neg P} \uparrow\downarrow notDef$$

$$\frac{}{\Gamma \vdash \mathsf{false} \Rightarrow P} falseDef$$

$$\frac{}{\Gamma \vdash P \Rightarrow \mathsf{true}} trueDef$$

$$\frac{\Gamma \vdash P \Rightarrow Q \land Q \Rightarrow P}{\Gamma \vdash P \Leftrightarrow Q} \uparrow\downarrow iffDef$$

F.3.4 Quantifier rules

$$\frac{\Gamma \ddagger S \vdash P}{\Gamma \vdash \forall S \bullet P} AllI$$

$$\frac{\Gamma \vdash \forall S \bullet P \qquad \Gamma \vdash b \in S}{\Gamma \vdash (\!| b |\!) \odot P} AllE$$

$$\frac{\Gamma \vdash (\!| b |\!) \odot P \qquad \Gamma \vdash b \in S}{\Gamma \vdash \exists S \bullet P} ExistsI$$

$$\frac{\Gamma \vdash \exists S \bullet P \qquad \Gamma \ddagger S \ddagger P \vdash Q}{\Gamma \vdash Q} ExistsE$$

Editor's note: UNIQUEQUANT ?

F.3.5 Expression rules

Sets

$$\frac{\Gamma \ddagger [x] T \vdash e = \{ (\!| x := s |\!) \odot T \bullet y \}}{\Gamma \ddagger [x] T \vdash y_{[s]} \in e} \uparrow\downarrow \begin{matrix} D2 \\ (y \in \alpha(\mathbf{wf}\, T)) \end{matrix}$$

$$\frac{\Gamma \vdash \begin{matrix} (\forall x : s \bullet x \in t) \land \\ (\forall x : t \bullet x \in s) \end{matrix}}{\Gamma \vdash s = t} \uparrow\downarrow Seteq(x \notin \phi s \cup \phi t)$$

$$\frac{\Gamma \vdash v = e_1 \lor \ldots \lor v = e_n}{\Gamma \vdash v \in \{e_1, \ldots, e_n\}} \uparrow\downarrow \mathit{Extmem}$$

$$\frac{\Gamma \vdash \exists S \bullet e = u}{\Gamma \vdash e \in \{S \bullet u\}} \uparrow\downarrow \mathit{Setcomp}(\phi u \cap \alpha S = \varnothing)$$

Cartesian products

$$\frac{\Gamma \vdash \forall x : t \bullet x \in s}{\Gamma \vdash t \in \mathbb{P} s} \uparrow\downarrow \mathit{Powerset}(x \notin \phi s)$$

$$\frac{\Gamma \vdash v = e_i}{\Gamma \vdash v = (e_1, \ldots, e_n).i} \mathit{Tupleequ}(1 \leq i \leq n)$$

$$\frac{\Gamma \vdash u.1 \in s_1 \land \ldots \land u.n \in s_n}{\Gamma \vdash u \in s_1 \times \cdots \times s_n} \uparrow\downarrow \mathit{Prodmem}$$

$$\frac{\Gamma \vdash u = (e_1, \ldots, e_n)}{\Gamma \vdash u.1 = e_1 \land \ldots \land u.n = e_n} \uparrow\downarrow \mathit{Tuplesel}$$

Labelled products

$$\frac{}{\Gamma \vdash \langle\!| x_1 := e_1, \ldots, x_n := e_n |\!\rangle .x_i = e_i} \begin{array}{l}\mathit{BindEqu} \\ (1 \leq i \leq n)\end{array}$$

$$\frac{\Gamma \vdash u.x_1 = e_1 \land \ldots \land u.x_n = e_n}{\Gamma \vdash u = \langle\!| x_1 := e_1, \ldots, x_n := e_n |\!\rangle} \uparrow\downarrow \mathit{BindSel}$$

$$\frac{}{\Gamma \ddagger \langle\!| b |\!\rangle \vdash x = b.x} D6(x \in \alpha b, \mathbf{wf}\, b)$$

Schemas

$$\frac{\Gamma \vdash e.x_1 = x_1 \land \cdots \land e.x_n = x_n \qquad \Gamma \vdash S}{\Gamma \vdash e = \theta S} D20$$

$$\frac{\vdash \langle\!| x_1 := x_1, \ldots, x_n := x_n |\!\rangle \in S}{\vdash S} \begin{array}{l}\uparrow\downarrow D8 \\ \alpha S = \{x_1, \ldots, x_n\}\end{array}$$

Description

$$\frac{\begin{array}{c}\Gamma \vdash (e,u) \in f \\ \Gamma \ddagger y : f \ \vdash \ (y.1 = e) \Rightarrow (y.2 = ey)\end{array}}{\Gamma \vdash u = f(e)}\ D18 \quad (y \notin \phi e \cup \phi u)$$

$$\frac{\begin{array}{c}\Gamma \vdash e \in s \\ \Gamma \vdash \langle\!| \ x := e \ |\!\rangle \odot P \\ \Gamma \ddagger y : s \vdash \langle\!| \ x := y \ |\!\rangle \ P \Rightarrow y = e\end{array}}{\Gamma \vdash e = \mu x : s \mid P}\ D19$$

Editor's note: IF THEN ELSE

Substitution

$$\frac{\Gamma \ddagger \langle\!| \ b \ |\!\rangle \ \vdash u = e}{\Gamma \vdash u = \langle\!| \ b \ |\!\rangle \circ e} \uparrow\downarrow D7b(\alpha b \cap \phi u = \varnothing)$$

$$\frac{\Gamma \vdash v = \langle\!| \ x := u \ |\!\rangle \circ e}{\Gamma \ddagger x := u \vdash v = e} \uparrow\downarrow Usedef(x \notin \phi v)$$

F.3.6 Schema calculus

$$\frac{\Gamma \vdash u \in [x_1 : s_1; \ldots; x_n : s_n]}{\Gamma \vdash u.x_1 \in s_1 \wedge \ldots \wedge u.x_n \in s_n} \uparrow\downarrow BindProd$$

$$\frac{\Gamma \vdash b \in S \qquad \vdash \langle\!| \ b \ |\!\rangle \odot P}{\Gamma \vdash b \in [S \mid P]} \uparrow\downarrow ?SchemaMem$$

$$\frac{\Gamma \vdash \langle\!| \ b \ |\!\rangle \odot S}{\Gamma \vdash b \in S} \uparrow\downarrow D14(\mathbf{wf}\, S)$$

$$\frac{\Gamma \vdash \langle\!| \ b \ |\!\rangle \odot [b \circ S]}{\Gamma \vdash b \in S} \uparrow\downarrow D15(\mathbf{wf}\, b)$$

$$\frac{\Gamma \vdash \neg \langle\!| \ b \ |\!\rangle \odot S}{\Gamma \vdash b \in \neg S} \uparrow\downarrow SNot(\mathbf{wf}\, S)$$

$$\frac{\Gamma \vdash \langle\!| \ b \ |\!\rangle \odot S \wedge \langle\!| \ b \ |\!\rangle \odot T}{\Gamma \vdash b \in (S \wedge T)} \uparrow\downarrow SAnd(\mathbf{wf}\, S \wedge T)$$

$$\frac{\Gamma \vdash \langle\!| b |\!\rangle \odot S \vee \langle\!| b |\!\rangle \odot T}{\Gamma \vdash b \in (S \vee T)} \uparrow\downarrow Sor(\mathbf{wf}\, S \vee T)$$

$$\frac{\Gamma \vdash \langle\!| b |\!\rangle \odot S \Rightarrow \langle\!| b |\!\rangle \odot T}{\Gamma \vdash b \in (S \Rightarrow T)} \uparrow\downarrow SImp(\mathbf{wf}\, S \Rightarrow T)$$

$$\frac{\Gamma \vdash \langle\!| b |\!\rangle \odot S \Leftrightarrow \langle\!| b |\!\rangle \odot T}{\Gamma \vdash b \in (S \Leftrightarrow T)} \uparrow\downarrow SIff(\mathbf{wf}\, S \Leftrightarrow T)$$

$$\frac{\Gamma \vdash \exists S \bullet \langle\!| b |\!\rangle \odot T}{\Gamma \vdash b \in \exists S \bullet T} \uparrow\downarrow SExists \quad (\phi T \cap (\alpha b \cup \alpha S)) = \varnothing$$

$$\frac{\Gamma \vdash \forall S \bullet \langle\!| b |\!\rangle \odot T}{\Gamma \vdash b \in \forall S \bullet T} \uparrow\downarrow SAll \quad (\phi T \cap (\alpha b \cup \alpha S)) = \varnothing$$

Editor's note: SHIDING, SPROJECTION, SCOMPOSITION, SDECORATION, SSUBSTITUTION, SUNIQUEQUANT

F.4 Type inference

F.4.1 Structural rules

$$\frac{\Gamma \vdash \Theta}{\Pi \ddagger \Gamma \vdash \Theta} \; T21$$

$$\frac{\Gamma \vdash \Theta}{\Gamma \ddagger \Pi \vdash \Theta} \; T22$$

$$\frac{\Gamma \vdash x \mathrel{:} \tau}{\Gamma \ddagger [y] \vdash x \mathrel{:} \tau} \; T23a$$

$$\frac{\Gamma \vdash x \mathrel{:} \tau}{\Gamma \ddagger y := e \vdash x \mathrel{:} \tau} \; T23b$$

$$\frac{\Gamma \vdash x \mathrel{:} \tau}{\Gamma \ddagger y : s \vdash x \mathrel{:} \tau} \; T23c$$

$$\frac{\Gamma \vdash y \mathrel{:} \tau}{\Gamma \ddagger [x] T \vdash y \mathrel{:} \tau} \; T25(y \notin \alpha[x]T)$$

F.4.2 Paragraphs

$$\frac{}{\Gamma \vdash [x] ::} \; T17$$

$$\frac{\Gamma \vdash s \mathrel{:} \mathcal{P}\tau}{\Gamma \vdash x : s ::} \; T18a$$

$$\frac{\Gamma \vdash e \mathrel{:} \tau}{\Gamma \vdash x := e ::} \; T18b$$

$$\frac{\Gamma \vdash P \;\surd}{\Gamma \vdash P ::} \; T19$$

$$\frac{\vdash \Gamma :: \qquad \Gamma \vdash \Pi ::}{\vdash \Gamma \ddagger \Pi \;\; ::} \; T20$$

$$\frac{\Gamma \ddagger [x] \vdash T ::}{\Gamma \vdash [x] T ::} \; T24(x \text{ not given set of } \Gamma)$$

$$\frac{\Gamma \vdash b \mathrel{:} \Sigma(x_1 \rightsquigarrow \tau_1, \ldots, x_n \rightsquigarrow \tau_n)}{\Gamma \vdash (\!|\, b \,|\!) \;\; ::} \; T30$$

$$\frac{\Gamma \vdash S \mathrel{:} \mathcal{P}\Sigma(x_1 \rightsquigarrow \tau_1, \ldots, x_n \rightsquigarrow \tau_n)}{\Gamma \vdash S ::} \; T34$$

F.4.3 Predicates

$$\frac{\Gamma \vdash e_1 \mathbin{:} \tau \qquad \Gamma \vdash e_2 \mathbin{:} \tau}{\Gamma \vdash e_1 = e_2 \ \surd} \ T1$$

$$\frac{\Gamma \vdash e \mathbin{:} \tau \qquad \Gamma \vdash s \mathbin{:} \mathcal{P}\tau}{\Gamma \vdash e \in s \ \surd} \ T2$$

$$\frac{}{\Gamma \vdash \mathsf{true} \ \surd} \ T3a$$

$$\frac{}{\Gamma \vdash \mathsf{false} \ \surd} \ T3b$$

$$\frac{\Gamma \vdash P \ \surd}{\Gamma \vdash \neg P \ \surd} \ T4\neg$$

$$\frac{\Gamma \vdash P \ \surd \qquad \Gamma \vdash Q \ \surd}{\Gamma \vdash P \wedge Q \ \surd} \ T4\wedge$$

$$\frac{\Gamma \vdash P \ \surd \qquad \Gamma \vdash Q \ \surd}{\Gamma \vdash P \vee Q \ \surd} \ T4\vee$$

$$\frac{\Gamma \vdash P \ \surd \qquad \Gamma \vdash Q \ \surd}{\Gamma \vdash P \Rightarrow Q \ \surd} \ T4\Rightarrow$$

$$\frac{\Gamma \vdash P \ \surd \qquad \Gamma \vdash Q \ \surd}{\Gamma \vdash P \Leftrightarrow Q \ \surd} \ T4\Leftrightarrow$$

$$\frac{\Gamma \vdash e \mathbin{:} \tau \qquad \Gamma \ddagger x := e \vdash P}{\Gamma \vdash (\!| x := e |\!) \odot P \ \surd} \ T6$$

$$\frac{\Gamma \vdash S :: \qquad \Gamma \ddagger S \vdash P \ \surd}{\Gamma \vdash \forall S \bullet P \ \surd} \ T36\forall$$

$$\frac{\Gamma \vdash S :: \qquad \Gamma \ddagger S \vdash P \ \surd}{\Gamma \vdash \exists S \bullet P \ \surd} \ T36\exists$$

$$\frac{\Gamma \vdash S :: \qquad \Gamma \ddagger S \vdash P \ \surd}{\Gamma \vdash \exists_1 S \bullet P \ \surd} \ T36\exists_1$$

$$\frac{\begin{array}{l}\Gamma \vdash S \mathbin{:} \mathcal{P}\Sigma(x_1 \rightsquigarrow \tau_1, \ldots, x_n \rightsquigarrow \tau_n) \\ \Gamma \vdash x_1 \mathbin{:} \tau_1 \quad \ldots \quad \Gamma \vdash x_n \mathbin{:} \tau_n\end{array}}{\Gamma \vdash S \ \surd} \ T33$$

$$\frac{\Gamma \ddagger (\!| b |\!) \vdash P \ \surd}{\Gamma \vdash (\!| b |\!) \odot P \ \surd} \ T32a$$

F.4.4 Expressions

$$\frac{\Gamma \vdash s \mathbin{:} \mathcal{P}\tau}{\Gamma \ddagger x : s \vdash x \mathbin{:} \tau} \; T7a$$

$$\frac{\Gamma \vdash e \mathbin{:} \tau}{\Gamma \ddagger x := e \vdash x \mathbin{:} \tau} \; T7b$$

Editor's note: NUMBERL, STRINGL

$$\frac{\Gamma \ddagger [x] T \vdash s \mathbin{:} \mathcal{P}\tau' \qquad \Gamma \ddagger [x] \ddagger T \vdash y \mathbin{:} \tau}{\Gamma \ddagger [x] T \vdash y_{[s]} \mathbin{:} \{\mho x \mapsto \tau'\}^T \tau} \; T26$$

$$\frac{}{\Gamma \ddagger [x] \vdash x \mathbin{:} \mathcal{P}(\mho x)} \; T8$$

$$\frac{\Gamma \vdash e_1 \mathbin{:} \tau \quad \ldots \quad \Gamma \vdash e_n \mathbin{:} \tau}{\Gamma \vdash \{e_1, \ldots, e_n\} \mathbin{:} \mathcal{P}\tau} \; T9$$

$$\frac{\Gamma \ddagger S \vdash e \mathbin{:} \tau}{\Gamma \vdash \{S \bullet e\} \mathbin{:} \mathcal{P}\tau} \; T37$$

$$\frac{\Gamma \vdash s \mathbin{:} \mathcal{P}\tau}{\Gamma \vdash \mathbb{P} s \mathbin{:} \mathcal{P}(\mathcal{P}\tau)} \; T11$$

$$\frac{\Gamma \vdash e_1 \mathbin{:} \tau_1 \quad \ldots \quad \Gamma \vdash e_n \mathbin{:} \tau_n}{\Gamma \vdash (e_1, \ldots, e_n) \mathbin{:} \chi(\tau_1, \ldots, \tau_n)} \; T13$$

$$\frac{\Gamma \vdash s_1 \mathbin{:} \mathcal{P}\tau_1 \quad \ldots \quad \Gamma \vdash s_n \mathbin{:} \mathcal{P}\tau_n}{\Gamma \vdash s_1 \times \cdots \times s_n \mathbin{:} \mathcal{P}\chi(\tau_1, \ldots, \tau_n)} \; T14$$

$$\frac{\Gamma \vdash e \mathbin{:} \chi(\tau_1, \ldots, \tau_i, \ldots, \tau_n)}{\Gamma \vdash e.i \mathbin{:} \tau_i} \; T15 (1 \leq i \leq n)$$

$$\frac{\Gamma \vdash e_1 \mathbin{:} \tau_1 \quad \cdots \quad \vdash e_n \mathbin{:} \tau_n}{\Gamma \vdash (\!| x_1 := e_1, \ldots, x_n := e_n |\!) \mathbin{:} \Sigma(x_1 \rightsquigarrow \tau_1, \ldots, x_n \rightsquigarrow \tau_n)} \; T27$$

$$\frac{\Gamma \vdash x_1 \mathbin{:} \tau_1 \quad \cdots \quad \Gamma \vdash x_n \mathbin{:} \tau_n}{\Gamma \vdash \theta S \mathbin{:} \Sigma(x_1 \rightsquigarrow \tau_1, \ldots, x_n \rightsquigarrow \tau_n)} \; T44 \; \alpha S = \{x_1, \ldots, x_n\}$$

$$\frac{\Gamma \vdash b \mathbin{:} \Sigma(x_1 \rightsquigarrow \tau_1, \ldots, x_n \rightsquigarrow \tau_n)}{\Gamma \vdash b.x_i \mathbin{:} \tau_i} \; T28(1 \leq i \leq n)$$

$$\frac{\Gamma \vdash f \mathbin{:} \mathcal{P}\chi(\tau', \tau) \quad \Gamma \vdash e \mathbin{:} \tau'}{\Gamma \vdash f(e) \mathbin{:} \tau} \; T42$$

$$\frac{\Gamma \vdash s \mathbin{:} \mathcal{P}\tau \quad x : s \vdash P \ \surd}{\Gamma \vdash \mu x : s \mid P \mathbin{:} \tau} \; T43$$

Editor's note: IF THEN ELSE

$$\frac{\Gamma \vdash b \mathbin{:} \Sigma(x_1 \rightsquigarrow \tau_1, \ldots, x_i \rightsquigarrow \tau_i, \ldots, x_n \rightsquigarrow \tau_n)}{\Gamma \ddagger \langle\!| \, b \, |\!\rangle \vdash x_i \mathbin{:} \tau_i} \; T31 \; (1 \leq i \leq n)$$

$$\frac{\Gamma \ddagger \langle\!| \, b \, |\!\rangle \vdash e \mathbin{:} \tau}{\Gamma \vdash b \odot e \mathbin{:} \tau} \; T32b$$

$$\frac{\Gamma \vdash S \mathbin{:} \mathcal{P}\Sigma(x_1 \rightsquigarrow \tau_1, \ldots, x_i \rightsquigarrow \tau_i, \ldots, x_n \rightsquigarrow \tau_n)}{\Gamma \ddagger S \vdash x_i \mathbin{:} \tau_i} \; T35 \; (1 \leq i \leq n)$$

$$\frac{\Gamma \ddagger x := u \vdash e \mathbin{:} \tau}{\Gamma \vdash \langle\!| \, x := u \, |\!\rangle \odot e \mathbin{:} \tau} \; T16(1 \leq i \leq n)$$

F.4.5 Schema

$$\frac{\Gamma \vdash s_1 \mathbin{:} \mathcal{P}\tau_1 \quad \cdots \quad \Gamma \vdash s_n \mathbin{:} \mathcal{P}\tau_n}{\Gamma \vdash x_1 : s_1; \cdots; x_n : s_n \mathbin{:} \mathcal{P}\Sigma(x_1 \rightsquigarrow \tau_1, \ldots, x_n \rightsquigarrow \tau_n)} \; T29$$

$$\frac{\Gamma \vdash S \mathbin{:} \tau \qquad S \vdash P \ \surd}{\Gamma \vdash S \mid P \mathbin{:} \tau} \; T38$$

$$\frac{\Gamma \vdash S \mathbin{:} \tau}{\Gamma \vdash \neg S \mathbin{:} \tau} \; T39$$

$$\frac{\Gamma \vdash S \mathbin{:} \mathcal{P}\Sigma\varrho \qquad \vdash T \mathbin{:} \mathcal{P}\Sigma\varsigma}{\Gamma \vdash S \wedge T \mathbin{:} \mathcal{P}\Sigma(\varrho \sqcup \varsigma)} \; T40\wedge$$

$$\frac{\Gamma \vdash S \mathbin{:} \mathcal{P}\Sigma\varrho \qquad \vdash T \mathbin{:} \mathcal{P}\Sigma\varsigma}{\Gamma \vdash S \vee T \mathbin{:} \mathcal{P}\Sigma(\varrho \sqcup \varsigma)} \; T40\vee$$

$$\frac{\Gamma \vdash S \mathbin{:} \mathcal{P}\Sigma\varrho \qquad \vdash T \mathbin{:} \mathcal{P}\Sigma\varsigma}{\Gamma \vdash S \Rightarrow T \mathbin{:} \mathcal{P}\Sigma(\varrho \sqcup \varsigma)} \; T40\Rightarrow$$

$$\frac{\Gamma \vdash S \mathbin{:} \mathcal{P}\Sigma\varrho \qquad \vdash T \mathbin{:} \mathcal{P}\Sigma\varsigma}{\Gamma \vdash S \Leftrightarrow T \mathbin{:} \mathcal{P}\Sigma(\varrho \sqcup \varsigma)} \; T40\Leftrightarrow$$

Editor's note: SPROJECTION , SHIDING

$$\frac{\Gamma \vdash S \mathbin{:} \mathcal{P}\Sigma\varsigma \qquad \vdash T \mathbin{:} \mathcal{P}\Sigma\varrho}{\Gamma \vdash \forall S \bullet T \mathbin{:} \mathcal{P}\Sigma(\varrho \setminus\setminus \varsigma)} \; T41\forall$$

$$\frac{\Gamma \vdash S \mathbin{:} \mathcal{P}\Sigma\varsigma \qquad \vdash T \mathbin{:} \mathcal{P}\Sigma\varrho}{\Gamma \vdash \exists S \bullet T \mathbin{:} \mathcal{P}\Sigma(\varrho \setminus\setminus \varsigma)} \; T41\exists$$

$$\frac{\Gamma \vdash S \mathbin{:} \mathcal{P}\Sigma\varsigma \qquad \vdash T \mathbin{:} \mathcal{P}\Sigma\varrho}{\Gamma \vdash \exists_1 S \bullet T \mathbin{:} \mathcal{P}\Sigma(\varrho \setminus\setminus \varsigma)} \; T41\exists_1$$

Editor's note: SCOMPOSITION, SDECORATION, SSUBSTITUTION

F.5 Free variables and alphabets

F.5.1 Paragraphs

$$
\begin{aligned}
\phi[x] &= \varnothing \\
\phi P &= \Phi P \\
\phi S &= \\
\phi(x : s) &= \phi s \\
\phi(x := e) &= \phi e \\
\phi([x]T) &= \phi T \setminus \{x\} \\
\phi(\Pi_1 \ddagger \Pi_2) &= \phi\Pi_1 \cup (\phi\Pi_2 \setminus \alpha\Pi_1)
\end{aligned}
$$

$$
\begin{aligned}
\alpha[x] &= \{x\} \\
\alpha P &= \varnothing \\
\alpha S &= \\
\alpha(x : s) &= \{x\} \\
\alpha(x := e) &= \{x\} \\
\alpha([x]T) &= \alpha T \\
\alpha(\Pi_1 \ddagger \Pi_2) &= \alpha\Pi_1 \cup \alpha\Pi_2
\end{aligned}
$$

F.5.2 Predicates

$$
\begin{aligned}
\Phi(e \in s) &= \phi e \cup \phi s \\
\Phi(e = v) &= \phi e \cup \phi v \\
\Phi\mathsf{true} &= \varnothing \\
\Phi\mathsf{false} &= \varnothing \\
\Phi(\neg P) &= \Phi P \\
\Phi(P \wedge Q) &= \Phi P \cup \Phi Q \\
\Phi(P \vee Q) &= \Phi P \cup \Phi Q \\
\Phi(P \Rightarrow Q) &= \Phi P \cup \Phi Q \\
\Phi(P \Leftrightarrow Q) &= \Phi P \cup \Phi Q \\
\Phi \forall S \bullet P &= \phi S \cup (\Phi P \setminus \alpha S) \\
\Phi \exists S \bullet P &= \phi S \cup (\Phi P \setminus \alpha S) \\
\Phi \exists_1 S \bullet P &= \phi S \cup (\Phi P \setminus \alpha S) \\
\Phi(S) &= \alpha S \cup \phi S \\
\Phi(\langle\!| x := e |\!\rangle \odot P) &= \phi e \cup (\Phi P \setminus \{x\}) \quad (?) \\
\Phi(\langle\!| b |\!\rangle \odot P) &= \Phi P \setminus \alpha b \quad (?)
\end{aligned}
$$

F.5.3 Schemas

$$
\begin{aligned}
\phi[x_1 : s_1; \cdots; x_n : s_n] &= \phi(s_1) \cup \cdots \cup \phi(s_n) \\
\phi[S \mid P] &= \phi S \cup (\Phi P \setminus \alpha S) \\
\phi(\neg S) &= \phi S \\
\phi(S \land T) &= \phi S \cup \phi T \\
\phi(S \lor T) &= \phi S \cup \phi T \\
\phi(S \Rightarrow T) &= \phi S \cup \phi T \\
\phi(S \Leftrightarrow T) &= \phi S \cup \phi T \\
\phi(S \mathbf{Proj} T) &= \\
\phi(S \backslash [x_1, \ldots, x_n]) &= \\
\phi(\forall S \bullet T) &= \phi S \cup \phi T \\
\phi(\exists S \bullet T) &= \phi S \cup \phi T \\
\phi(\exists_1 S \bullet T) &= \phi S \cup \phi T \\
\phi(S[x_1/y_1, \ldots, x_n/y_n]) &= \\
\phi(S \fatsemi T) &= \\
\phi(S^q) &= \\
\phi(b \circ S) &=
\end{aligned}
$$

$$
\begin{aligned}
\alpha[x_1 : s_1, \ldots, x_n : s_n] &= \{x_1, \ldots, x_n\} \\
\alpha[S \mid P] &= \alpha S \\
\alpha(\neg S) &= \alpha S \\
\alpha(S \land T) &= \alpha S \cup \alpha T \\
\alpha(S \lor T) &= \alpha S \cup \alpha T \\
\alpha(S \Rightarrow T) &= \alpha S \cup \alpha T \\
\alpha(S \Leftrightarrow T) &= \alpha S \cup \alpha T \\
\alpha(S \mathbf{Proj} T) &= \\
\alpha(S \backslash [x_1, \ldots, x_n]) &= \\
\alpha(\forall S \bullet T) &= \alpha T \setminus \alpha S \\
\alpha(\exists S \bullet T) &= \alpha T \setminus \alpha S \\
\alpha(\exists_1 S \bullet T) &= \alpha T \setminus \alpha S \\
\alpha(S[x_1/y_1, \ldots, x_n/y_n]) &= \\
\alpha(S \fatsemi T) &= \\
\alpha(S^q) &= \\
\alpha(b \circ S) &= \\
\alpha\langle\!| x_1 := e_1, \ldots, x_n := e_n |\!\rangle &= \{x_1, \ldots, x_n\}
\end{aligned}
$$

F.5.4 Expressions

$$
\begin{aligned}
\phi(x) &= \{x\} \\
\phi(x[y]) &= \\
\phi(i) &= \\
\phi(z) &= \\
\phi\{e_1, \ldots, e_n\} &= \phi(e_1) \cup \cdots \cup \phi(e_n) \\
\phi\{S \bullet e\} &= \phi S \cup (\phi e \setminus \alpha S) \\
\phi(\mathbb{P}\ s) &= \phi(s) \\
\phi(e_1, \ldots, e_n) &= \phi(e_1) \cup \cdots \cup \phi(e_n) \\
\phi(s_1 \times \cdots \times s_n) &= \phi(s_1) \cup \cdots \cup \phi(s_n) \\
\phi(e.i) &= \phi(e) \\
\phi(\!| x_1 := e_1, \ldots, x_n := e_n |\!) &= \phi(e_1) \cup \cdots \cup \phi(e_n) \\
\phi(\theta S) &= \Phi S \\
\phi(b.x) &= \phi(b) \quad (?) \\
\phi(f(e)) &= \phi f \cup \phi e \\
\phi(\mu\, S \bullet e) &= \phi S \cup \Phi(e_1 \setminus \alpha S) \\
\phi(\textbf{if } P \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ fi}) &= \\
\phi(b \ominus e) &= \phi e \setminus \alpha b \quad (?) \\
\phi(\!| x := u |\!) \ominus e) &= \phi(u) \cup \phi(e) \setminus \{x\} \quad (?)
\end{aligned}
$$

F.6 Substitution

F.6.1 Predicates

$$
\begin{aligned}
b \odot (e = u) &\equiv b \circ e = b \circ u \\
b \odot (e \in s) &\equiv b \circ e \in b \circ u \\
b \odot \mathsf{true} &\equiv \mathsf{true} \\
b \odot \mathsf{false} &\equiv \mathsf{false} \\
b \odot \neg P &\equiv \neg\, b \odot P \\
b \odot (P \land Q) &\equiv b \odot P \land b \odot Q \\
b \odot (P \lor Q) &\equiv b \odot P \lor b \odot Q \\
b \odot (P \Rightarrow Q) &\equiv b \odot P \Rightarrow b \odot Q \\
b \odot (P \Leftrightarrow Q) &\equiv b \odot P \Leftrightarrow b \odot Q
\end{aligned}
$$

When $\alpha b \cap \Phi P \subseteq \alpha S$:

$$
\begin{aligned}
b \odot \forall S \bullet P &\equiv \forall b \circ S \bullet P \\
b \odot \exists S \bullet P &\equiv \exists b \circ S \bullet P \\
b \odot \exists_1 S \bullet P &\equiv \exists_1 b \circ S \bullet P
\end{aligned}
$$

When $\alpha b \cap \alpha S \cap \Phi P = \varnothing$ and $\alpha S \cap \phi b = \varnothing$:

$$
\begin{aligned}
b \odot \forall S \bullet P &\equiv \forall b \circ S \bullet b \odot P \\
b \odot \exists S \bullet P &\equiv \exists b \circ S \bullet b \odot P \\
b \odot \exists_1 S \bullet P &\equiv \exists_1 b \circ S \bullet b \odot P
\end{aligned}
$$

When $\alpha b \cap \alpha S = \varnothing$:

$$
b \odot S \equiv [b \circ S]
$$

When $\mathbf{wf}\, b$:

$$
b \odot S \equiv b \odot [b \circ S]
$$

$$
\begin{aligned}
\langle\!|\, y := v \,|\!\rangle \odot (\langle\!|\, y := u \,|\!\rangle\ P) &\equiv \langle\!|\, y := \langle\!|\, y := v \,|\!\rangle \circ u \,|\!\rangle \odot P \\
\langle\!|\, x := v \,|\!\rangle \odot (\langle\!|\, y := u \,|\!\rangle\ P) &\equiv \\
\langle\!|\, y := \langle\!|\, x := v \,|\!\rangle \circ u \,|\!\rangle \odot (\langle\!|\, x := v \,|\!\rangle \odot P) & \\
\text{where } y \notin \phi v &
\end{aligned}
$$

F.6.2 Schema predicates

$$
\begin{aligned}
b \odot [S \mid P] &\equiv b \odot S \wedge b \odot P \\
b \odot [\neg S] &\equiv \neg b \odot S \\
b \odot [S \wedge T] &\equiv b \odot S \wedge b \odot T \\
b \odot [S \vee T] &\equiv b \odot S \vee b \odot T \\
b \odot [S \Rightarrow T] &\equiv b \odot S \Rightarrow b \odot T \\
b \odot [S \Leftrightarrow T] &\equiv b \odot S \Leftrightarrow b \odot T \\
b \odot [S \ \mathbf{Proj}\ T] &\equiv \\
b \odot S \backslash [x_1, \ldots, x_n] &\equiv \\
\text{When } \alpha S \cap (\phi b \odot T \cup \alpha b) = \varnothing&\text{:} \\
b \odot [\forall S \bullet T] &\equiv \forall b \odot S \bullet b \odot T \\
b \odot [\exists S \bullet T] &\equiv \exists b \odot S \bullet b \odot T \\
b \odot [\exists_1 S \bullet T] &\equiv \exists_1 b \odot S \bullet b \odot T \\
b \odot [S[x_1/y_1, \ldots, x_n/y_n]] &\equiv \\
b \odot [S \fatsemi T] &\equiv \\
b \odot [S^?] &\equiv \\
b_1 \odot [b \odot S] &\equiv
\end{aligned}
$$

F.6.3 Expressions

$$
\begin{aligned}
b \odot x &\equiv b.x \quad \text{when } x \in \alpha b \\
b \odot x &\equiv x \quad \text{when } x \notin \alpha b \\
b \odot x[y] &\equiv \\
b \odot i &\equiv \\
b \odot \{e_1, \ldots, e_n\} &\equiv \{b \odot e_1, \ldots, b \odot e_n\} \\
\text{When } \alpha b \cap \phi e \subseteq \alpha S&\text{:} \\
b \odot \{S \bullet e\} &\equiv \{b \odot S \bullet e\} \\
\text{When } \alpha b \cap \alpha S \cap \phi e = \varnothing &\text{ and } \alpha S \cap \phi b = \varnothing\text{:} \\
b \odot \{S \bullet e\} &\equiv \{b \odot S \bullet b \odot e\} \\
b \odot \mathbb{P}\ s &\equiv \mathbb{P}\, b \odot s \\
b \odot ((e_1, \ldots, e_n)) &\equiv (b \odot e_1, \ldots, b \odot e_n) \\
b \odot (s_1 \times \cdots \times s_n) &\equiv b \odot s_1 \times \cdots \times b \odot s_n \\
b \odot (e.i) &\equiv (b \odot e).i \\
b \odot (\!| x_1 := e_1, \ldots, x_n := e_n |\!) &\equiv \\
&\quad (\!| x_1 := b \odot e_1, \ldots, x_n := b \odot e_n |\!) \\
b \odot \theta S &\equiv \theta b \odot S \\
&\quad \text{when } \alpha b \cap \alpha S = \varnothing \\
b \odot \theta S &\equiv b \\
&\quad \text{when } \alpha b = \alpha S \\
b_1 \odot b.x &\equiv (b_1 \odot b).x \\
b \odot (f(e)) &\equiv (b \odot f)(b \odot e) \\
b \odot (\mu\, S \bullet e) &\equiv \text{Agh. Look at Stephen's Fig 9.1} \\
b \odot (\textbf{if } P \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ fi}) &\equiv \\
b_1 \odot (b \odot e) &\equiv \\
(\!| x := v |\!) \odot ((\!| x := u |\!) \odot e) &\equiv (\!| x := (\!| x := v |\!) \odot u |\!) \odot e \\
(\!| y := v |\!) \odot ((\!| x := u |\!) \odot e) &\equiv (\!| x := (\!| y := v |\!) \odot u |\!) \odot \\
&\qquad\qquad ((\!| y := v |\!) \odot e) \\
&\quad \text{when } x \notin \phi v.
\end{aligned}
$$

F.6.4 Schema expressions

$$
\begin{aligned}
b\odot[S \mid P] &\equiv [b\odot S \mid b \odot P] \\
&\quad \text{when } \alpha b \cap \alpha S = \varnothing \\
b\odot[S \mid P] &\equiv [b\odot S \mid P] \\
&\quad \text{when } \alpha b \cap \Phi P \subseteq \alpha S \\
b\odot[\neg S] &\equiv [\neg b\odot S] \\
b\odot[S \wedge T] &\equiv [b\odot S \wedge b\odot T] \\
b\odot[S \vee T] &\equiv [b\odot S \vee b\odot T] \\
b\odot[S \Rightarrow T] &\equiv [b\odot S \Rightarrow b\odot T] \\
b\odot[S \Leftrightarrow T] &\equiv [b\odot S \Leftrightarrow b\odot T] \\
b\odot[S \mathbf{Proj} T] &\equiv \\
b\odot S\backslash[x_1, \ldots, x_n] &\equiv \\
b\odot[\forall S \bullet T] &\equiv [\forall b\odot S \bullet b\odot T] \\
b\odot[\exists S \bullet T] &\equiv [\exists b\odot S \bullet b\odot T] \\
b\odot[\exists_1 S \bullet T] &\equiv [\exists_1 b\odot S \bullet b\odot T] \\
b\odot[S[x_1/y_1, \ldots, x_n/y_n]] &\equiv \\
b\odot[S \fatsemi T] &\equiv \\
b\odot[S^q] &\equiv \\
b_1\odot[b\odot S] &\equiv
\end{aligned}
$$

F.7 Provisos as judgements

$$\frac{\Gamma \vdash P}{\Gamma' \vdash P'}\ (\alpha S = s) \equiv \frac{\Gamma \vdash P \qquad \Gamma' \vdash \alpha S = s}{\Gamma' \vdash P'}$$

$$\frac{\Gamma \vdash \phi S = x \qquad \Gamma \vdash \alpha S = z \qquad \Gamma \ddagger S \vdash \phi P = y}{\Gamma \vdash \phi \forall S \bullet P = x \cup (y \setminus z)}$$

□

G References – Informative Annex

Notes on this section of the Z Standard

Section title: References
Section editor: John Nicholls
Source file: infref.tex
Most recent update: 29th June 1995
Formatted: 3rd July 1995

References

[1] Abrial, J-R., "A Course on System Specification," Lecture Notes, Programming Research Group, University of Oxford, 1981.

[2] Bowen, J. P., "Select Z Bibliography," available from newsgroup `comp.specification.z`.

[3] Brien, S.M., Gardiner, P.H.B., Lupton, P.J., Woodcock, J.C.P., "A Semantics for Z," in preparation, 1992.

[4] Brien, S.M., Nicholls, J.E., *Z Base Standard* Version 1.0, Working Draft of the Z Standards Panel. Also published as Technical Monograph PRG-107, Oxford University Computing Laboratory, 1992.

[5] BSI Standard **BS 0** : Part 1 : 1981, *A standard for standards. Part 1. Guide to general principles of standardization*, British Standards Institution, 1991.

[6] BSI Standard **BS 6154**, *Method of defining syntactic metalanguage*, British Standards Institution, 1981.

[7] Enderton, H.B., *Elements of set theory*, Academic Press, 1977.

[8] Gardiner, P.H.B., Lupton, P.J., Woodcock, J.C.P., "A simpler semantics for Z," in J. E. Nicholls (ed), *Z User Workshop, Oxford 1990*, Proceedings of the Fifth Annual Z User Meeting, Springer-Verlag, 1991.

[9] Goldfarb, C. F., *The SGML Handbook*, Clarendon Press, Oxford, 1990.

[10] Hamilton, A.G., *Numbers, sets and axioms*, Cambridge University Press, 1982.

[11] Hayes, I. J., (ed.), *Specification Case Studies*, Prentice-Hall International, 1987.

[12] ISO (International Organization for Standardization), **ISO 8879-1986 (E)** *Information Processing – Text and Office systems – Standard Generalized Markup Language (SGML)*, Geneva: ISO, 1986.

[13] Jones, C. B., *Software Development–A Rigorous Approach*, Prentice-Hall International, 1980.

[14] Jones, C. B., *Systematic Software Development using VDM*, Prentice-Hall International, 1986.

[15] King, S., Sørensen, I. H., Woodcock, J.C.P., "Z: Grammar and Concrete and Abstract Syntaxes (Version 2.0)," Technical Monograph PRG-68, Programming Research Group, University of Oxford, 1988.

[16] Lalonde, W.R., Des Rivieres, J., "Handling Operator Precedence in Arithmetic Expressions with Tree Transformations," *ACM Transactions on Programming Languages and Systems*, Vol 3, No 1, January 1981.

[17] McMorran, M.A., Nicholls, J.E., " Z User Manual," Technical Report TR12.274, IBM Hursley Park, 1989.

[18] Morgan, C. C., "Schemas in Z: a Preliminary Reference Manual," Programming Research Group, University of Oxford, 1984.

[19] Nicholls, J.E., "Domains of application for formal methods," in *Proceedings of Z User Workshop*, University of York, *Workshops in Computing*, Springer-Verlag, 1992.

[20] Sennett, C. T., "Syntax and Lexis of the Specification Language Z," RSRE Memorandum No. 4367, 1990.

[21] Sørensen, I. H., "A Specification Language," in *Program Specification* (J. Staunrup, ed.), Lecture Notes in Computer Science, vol. 134, Springer-Verlag, 1982.

[22] Sperberg-McQueen, C.M., Burnard, Lou (editors), *Guidelines for Electronic Text Encoding and Interchange*, TEI P3, Text Encoding Initiative, Chicago, Oxford April 8, 1994.

[23] Spivey, J. M., *Understanding Z: a specification language and its formal semantics*, Cambridge University Press, 1988.

[24] Spivey, J. M., *The Z notation – a reference manual*, Prentice-Hall International, 1989. *(2nd edition, 1992).*

[25] Stoy, J.E., *Denotational semantics: the Scott-Strachey approach to programming language theory*, MIT Press, 1977.

[26] Sufrin, B. A., "Formal Specification: Notation and Examples," in *Tools and Notations for Program Construction* (D. Néel, ed.). Cambridge University Press, 1981.

[27] Sufrin, B. A., (ed.), "Z Handbook, Draft 1.1," Programming Research Group, University of Oxford, 1986.

[28] Woodcock, J. C. P., "Structuring Specifications in Z," *Software Engineering Journal*, Vol 4, No 1 (January 1989).

[29] Woodcock, J.C.P., Brien, S.M., "$\mathcal{W}$: a logic for Z," in J. E. Nicholls (ed), *Z User Workshop, York 1991*, Proceedings of the Sixth Annual Z User Meeting, Springer-Verlag, 1992.

□